Database Systems Implementation

Candidate Number: 1001689

Question 1 (a)

Introduction

A cool story about Factorized Representation

The first question is concerned with making less painful the conjunctive query R1(A,B), R2(A, C), by storing a cached copy of its result.
Our aim is to give our best in providing you a collection algorithms for incremental maintenance that are agnostic of the representation of the input tables and of the join algorithm, the idea is that we are always going to beat the naïve approach of rebuilding the result of a query from scratch.

Let's imagine we have the following result of the above join:

A	B	C
1	2	3
1	2	5
1	3	3
1	3	5

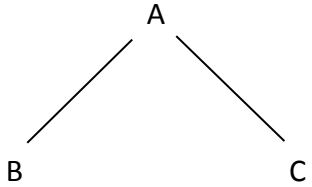
To be more precise each record can be represented as a product of singleton relations and the whole result can be represented as their respective union:

$$\langle 1 \rangle \times \langle 2 \rangle \times \langle 3 \rangle \cup$$
$$\langle 1 \rangle \times \langle 2 \rangle \times \langle 5 \rangle \cup$$
$$\langle 1 \rangle \times \langle 3 \rangle \times \langle 3 \rangle \cup$$
$$\langle 1 \rangle \times \langle 3 \rangle \times \langle 5 \rangle$$

We can then obtain a more compact representation with some algebraic manipulation by exploiting the distributive property of the product over the union allowing us to obtain the following:

$$\langle 1 \rangle \times (\langle 2 \rangle \cup \langle 3 \rangle) \times (\langle 2 \rangle \cup \langle 3 \rangle)$$

The above idea lies at the root of this work[1] and we are going to use this approach for representing the materialized view of the above conjunctive query, more specifically our conjunctive query's result will be represented with a factorization-tree that can be consumed:



[1] Bakibayev, Nurzhan, Dan Olteanu, and Jakub Závodný. "Fdb: A query engine for factorised relational databases." *Proceedings of the VLDB Endowment* 5.11 (2012): 1232-1243.

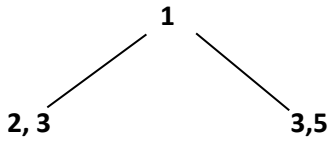
The tuples of the result can be obtained in the following way[1]:

$$\bigcup_{a \in A}(\langle a \rangle \times (\bigcup_{b \in B}\langle b \rangle \times \bigcup_{c \in C}\langle c \rangle)$$

In literature factorization-trees have been widely used for representing intermediate results of a query, those provide an alternative and succinct representation for the result of a query and perform better than standard compression algorithms (ZIP; LZM, etc.).

We now feel necessary giving you an example of what the above result represented through a factorization can look like, we feel that the example should give a general overview of the compressing capabilities of a factorization-tree (also shortly denoted as f-tree).

A	B	C
1	2	3
1	2	5
1	3	3
1	3	5



In general it is not always given that we can factorize a relation but we can always do that for the result of the join queries as we are always able to build a tree which internal nodes represent the joining attributes while the leafs represent the non-joining attributes.

Also f-trees speed up notably count queries due to its compressing nature.

For solving this first task other than f-trees we are going to employ a pair of tricks.

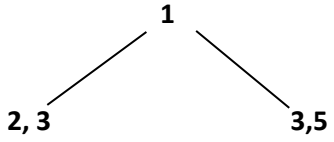
Under inserts

We note that if we we insert a new record *t* into table R1 the following about the updated query result holds:

$$(R_1 \cup t) \bowtie R_2 = R_1 \bowtie R_2 \cup (t_{R_1} \bowtie R_2)$$

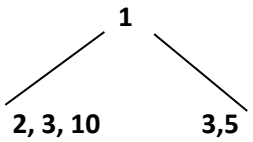
In a short given an already existing result we can obtain the new entries to add in the materialized view by joining the newly inserted tuple *t* with the other relations involved in the above conjunctive query excluding the one in which we inserted the record, in a nutshell one of the relations involved in the join has a single record which is a big bonus over the naïve approach to recompute the whole query result from scratch.
Also, assuming that all the tables must have a joining attribute (ie. an attribute that appears in other tables as well), we can employ another trick: so let's imagine we have the above below materialized view of the conjunctive query and that we are inserting the record (1, 10) in the table R1:

A	B	C
1	2	3
1	2	5
1	3	3
1	3	5



For obtaining the new entries to add we simply execute the join $(1, 10)_{R_1} \bowtie R_2$ but sometimes we can avoid that if we know that in the factorization-tree of the result there is already an internal node having the same value of the joining attribute of the new record thus obtaining:

A	B	C
1	2	3
1	2	5
1	3	3
1	3	5
1	10	3
1	10	5



If we could not find in the factorization-tree an internal node having the same value of the joining attribute we would have needed to run the query $(1, 10)_{R_1} \bowtie R_2$ but this is not the case.

We are going to give you another example, imagine the relations R1 and R2:

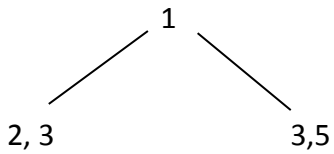
R1

A	B
1	2
1	3
2	8

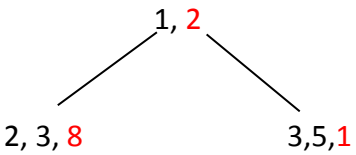
R2

A	C
1	3
1	5

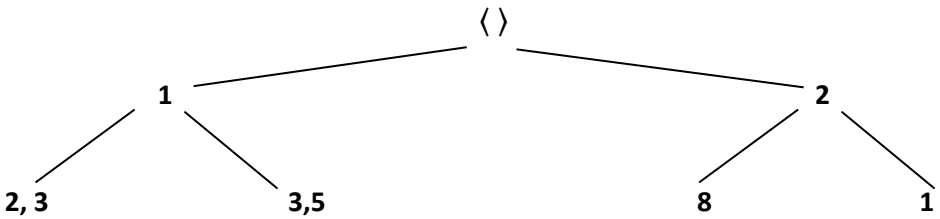
Thus the current factorization-tree of the result of the above join is still the same as before:



Adding the record $(2, 1)_{R_2}$, would require us to run the (small) join query $(2, 1)_{R_2} \bowtie R_1$ because we cannot find an internal node with value 2 in the factorization-tree, what we obtain at the end is the following:



The values with the same colour are part of the same tuple in the result.
We can think of the above also as (⟨ ⟩ denotes an empty relation):



Algorithm for updating the materialized view upon insertion of a new record

Function UpdateOnInsert(Factorization-Tree, Relation, Record):

(1) Let *t* be the value of the joining attribute of the Record
(2) If there is an internal node in the Factorization-Tree with value *t*:
    a. Simply attach the value of the non-joining attribute in the correct leaf of the sub-tree with root *t*
(3) Otherwise:
    a. Run the join with the record and all the relations except the one in which we inserted the record
    b. Create a subtree with value *t* as root denoted as *s*
    c. For each row in the result
        1. Attach the values of the non-joining attributes of the row to the leafs of *s*
    d. Attach the subtree to the root ⟨ ⟩

Our algorithm performs better than NaiveDB we note that in our worst-case scenario we perform a more lightweight join algorithm and that the cost of creating and updating the binary tree is indeed small, also we use tricks that are agnostic of the chosen join algorithm and of the representation of the input table.

Our implementation uses hash-join, and suppose we are inserting a record in R1 we can give precise asymptotic cost of the above algorithm being:

$$O(\ 2 *\ R_2\ )$$

That becomes then:

$$O(R_2)$$

So the complexity of the above is mainly driven by the cost of the join algorithm on a single record.

The naïve approach using sort-merge join has complexity:

$$O(\ |R_1| * \log(|R_1|) + |R_2| * \log(|R_2|) + |R_1 * R_2|)$$

Compare that with the above!

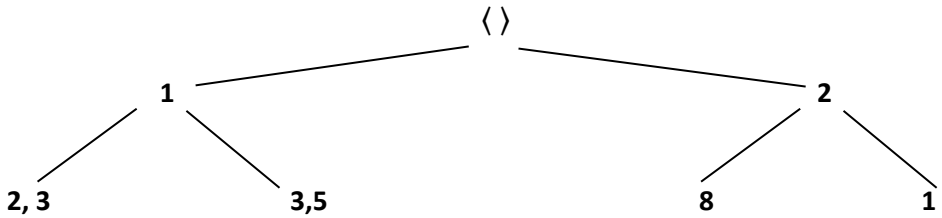
Now let's imagine that we wanted to be fairer and let NaiveDB use the same join method as the our Hash-Join, then the complexity becomes then:

$$O(R_1 + R_2)$$

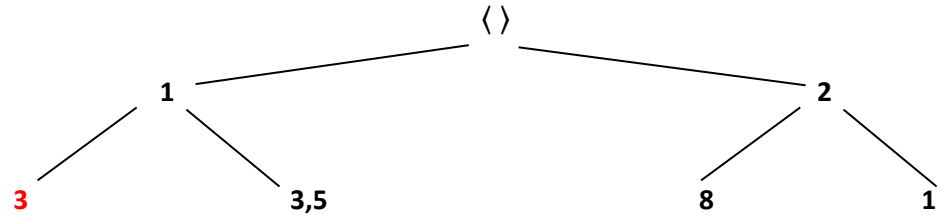
We still outperform NaiveDB from asymptotic standpoint asour tricks are agnostic of the chosen Join Method, the reason is simple: in the worst-case scenario we are going to run a smaller join compared to NaiveDB.

Under deletions

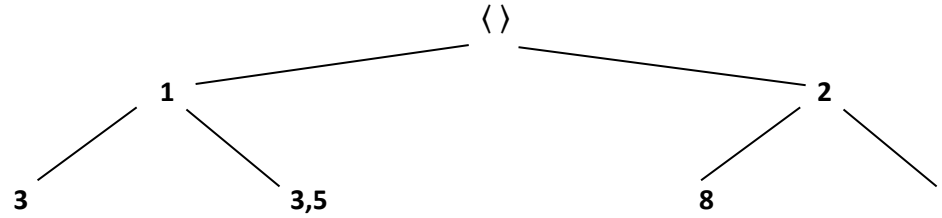
Let's imagine we have the following factorization-tree of the current result:



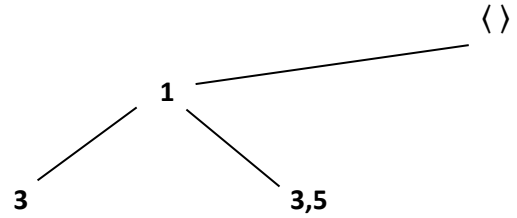
Suppose we are deleting the tuple $(1, 2)_{R_1}$ we now obtain the following:



So in general if we are deleting a tuple which joining attribute was in the result we simply empty that from the correct leaf.
Now suppose we delete $(2, 1)_{R_2}$ so we obtain:



We must remember to ALWAYS delete an internal node when one of its children gets emptied, thus from the above delete what we obtain is:



The rules for updating the factorization-tree after a deletion are:

1. Given the appropriate sub-tree delete the value residing at the leaf
2. If one of the leafs of the sub-tree is empty delete the parent as-well

The algorithm follows:

Algorithm for updating the materialized view upon deletion of a record

Function UpdateOnDelete(Factorization-Tree, Relation, Record):

(1) Let *t* be the value of the joining attribute of the Record
(2) If there is an internal node in the Factorization-Tree with value *t*:
    a. Delete the value of the non-joining attribute of the record from the leaf if there exists
    b. If we have emptied the leaf delete the sub-tree with root *t* as well
(3) Otherwise:
    a. We are done as either the record does not exists or it is not in the result so we do not care.

The complexity of the above algorithm is constant:

$$O(1)$$

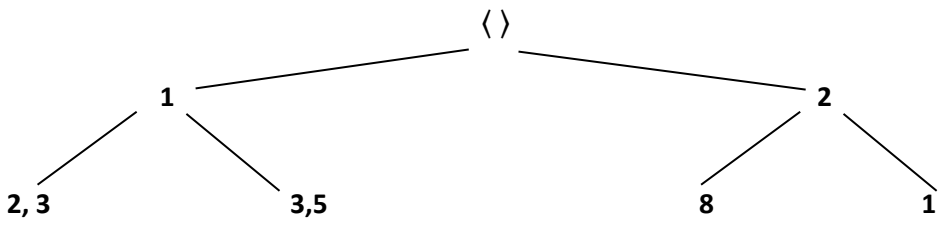
Count queries

In this section we are concerned in optimizing the query COUNT[R1(A,B), R2(A,C)].

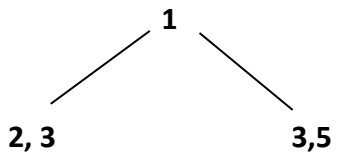
A note: from the exam sheet's paper we could not understand whether we were allowed to cache the result of count queries so we assumed we were not, we are going to propose an approach for count queries that in the worst-case scenario is equivalent to the naïve approach of counting by iterating over each row of the result, we do that by showing an upper bound of the size of the factorization-tree.

In this section we are going to extend the line of work we have pursued for incremental maintenance of the result of the query R1(A,B),R2(A,C) with factorization-trees, more specifically we are going to show an approach for doing counting over a factorization-tree.

A	B	C
1	2	3
1	2	5
1	3	3
1	3	5
2	8	1



The above factorization-tree represents a result with 5 rows, but before diving straight into the approach let's start with a smaller example of the sub-tree with joining value 1:



If we unfold the sub-tree we can obtain the rows of the result:

1	2	3
1	2	5
1	3	3
1	3	5

And then after unfolding that we can count the number of rows by iterating over each row and incrementing the counter, it turns out that we do not need to unfold the sub-tree, let's start with defining the operations:

- $COUNT - LEFT - LEAF(x)$
  The number of values in the left leaf of the subtree with value *x*
- $COUNT - RIGHT - LEAF(x)$
  The number of values in the right leaf of the subtree with value *x*

Now coming back to the subtree before note that:

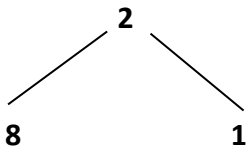
$$COUNT - LEAF - RIGHT(1) * COUNT - LEAF - LEFT(1) = 2 * 2 = 4$$

That is neat! We obtained the number of rows in the sub-tree without having to unfold the whole subtree.

Now we can define the following:

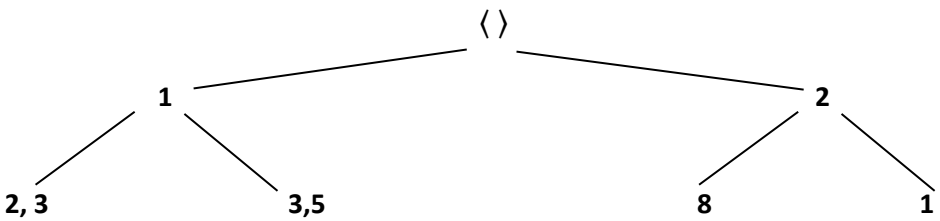
$$COUNT - NODE(x) = COUNT - LEAF - RIGHT(x) * COUNT - LEAF - LEFT(x)$$

So now let's move on the next subtree:



$$COUNT - NODE(2) =$$
$$COUNT - LEAF - RIGHT(2) * COUNT - LEAF - LEFT(2) = 1$$

Now we if we sum the count obtained for each internal node in the factorization tree we obtain the total number of results available in the factorization-tree:



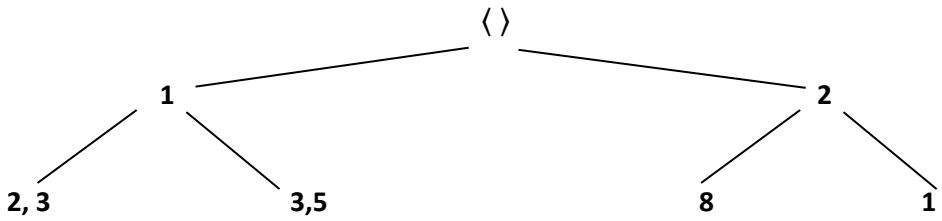
$$\#result = COUNT - NODE(1) + COUNT - NODE(2) = 5$$

Let's denote with *f* every internal node of the factorization-tree (in the previous example are 1 and 2), we can then deduct the following:

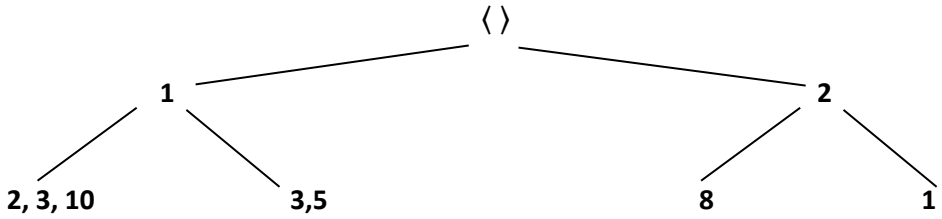
$$\#result = \sum_{f} COUNT - NODE(f)$$

So for counting queries we simply extend the algorithms for insertion and deletion previously described for the conjunctive query to include the extra step of counting over a factorization-tree that we have described.

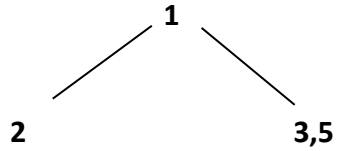
We note that if the insertion of a new result tuple involves the creation of a new subtree in the result then the count of the result is simply increased by the value of $COUNT - NODE$ otherwise let's imagine we have this factorization-tree:



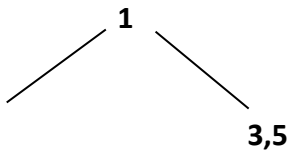
Imagine that we are adding the record $(1, 10)_{R_1}$ thus obtaining:



We have appended a value in the left-leaf of the subtree with root 1 so what is the new result of the subsequent count query? In this case it is simply increase by $COUNT - RIGH - LEAF(1)$ that is 2, *more precisely we increase the number of rows in the result by the number of the elements in the other leaf not affected by the insertion.*
The same thing applies for deletion, the number of deleted entries in the result is the count of the elements in the other leaf not affected by the deletion, let's imagine we have this factorization-tree:



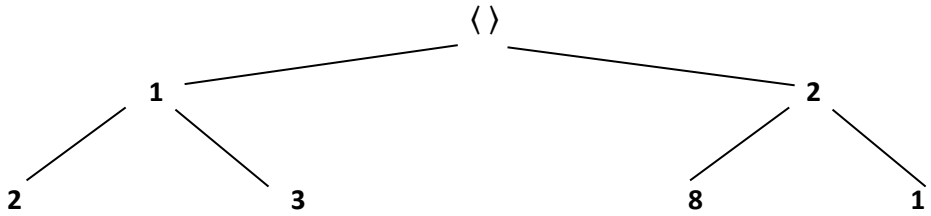
Now imagine we are deleting $(1,2)_{R_1}$ so we obtain:



So for obtaining the total number of deleted entries in the result we simply take the total number of the elements in the opposing leaf before checking if we need to remove this subtree from the factorization-tree which in our case we do because we have emptied the left leaf.

So in a nutshell we showed that it is not necessary to unfold the factorization-tree to obtain the count of the query and that instead we can operate directly on the factorization-tree and that it is possible to do incremental maintenance in the case we wished to do that for count queries.

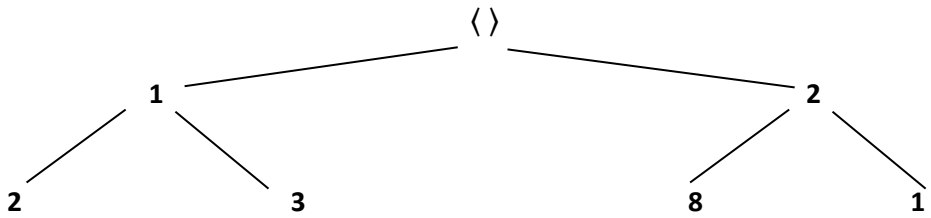
Why is this better than NaiveDB? Simply because that the worst case scenario is to have a factorization-tree linear in the number of the results (ie. with every subtree of the factorization-tree having each leaf filled with a single value)



In this scenario our approach is equivalent to iterating over each row of the result as NaiveDB would do but this is a worst-case scenario, also we can show that in the worst-case the factorization-tree size is linear in the size of the input database, thus the more our factorization-tree is compressed the more our count queries are speeded up compared to iterating over each row of the result.
We will be proving the upper bound of the size of the factorization-tree by using a known result: for any query q and a factorization-tree t there exists a rational number $s(t)$ such that there exists a factorization of the query q for a given database D with size $O(|D|^{s(t)})$.
Without going too much in detail (for which we refer to the work[2] and the helpful slides[3]) $s(t)$ is the maximum number of the relations covered for each node, for the factorization-tree chosen for representing the result of our conjunctive query this number is 1, hence our factorization-tree for the conjunctive query R1(A,B),R2(A,C) has size $O(|D|)$.

[2] Bakibayev, Nurzhan, Dan Olteanu, and Jakub Závodný. "Fdb: A query engine for factorised relational databases." *Proceedings of the VLDB Endowment* 5.11 (2012): 1232-1243.
[3] Dan Olteanu, Jakub Závodný, Factorised Relational Databases, Research Seminar, DCSIS, Birkbeck College, Slide

Data structures



We opted for a structure of this kind:

```
unordered_map<int, FactorizationTree_SubTree> factorizationTree;

struct FactorizationTree_SubTree {
	set<int> left_leaf;
	set<int> right_leaf;
};
```

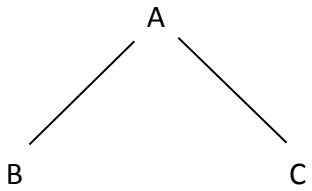
The idea is to have a hash-table for a mapping from the value of a joining attribute to the corresponding sub-tree and the leaf in the subtree are implemented using any data-structure that provide constant-cost membership testing and insertion emulating a set.

The input table can be represented as simple array of tuples consisting of two Integers as we provided algorithms that are agnostic of the chosen data-structure for the input-table and of the join-algorithms, the only constraint is that checking the existence of a tuple should be a constant operation so to make duplicates checking not expensive.

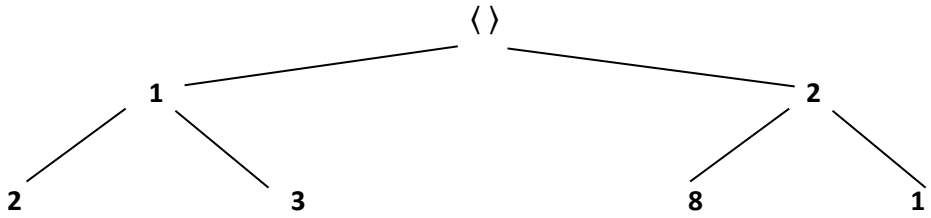
Question 1 (b)

This question is concerned with incremental maintenance of arbitrary conjunctive-queries over arbitrary databases.

Unfortunately as neat are the factorization-trees it turns out that it can be very challenging to maintain it for cyclic conjunctive queries (ie. R1(A,B),R2(A,C),R3(B,C)).
Suppose we decide to use the same factorization-tree as we did in the first task (as the result of both the queries have the same attributes) for the conjunctive query R1(A,B),R2(A,C),R3(B,C).:



Let's imagine we want to delete from the table R3 the tuple $(2,3)_{R_3}$ from the below instance of the above factorization tree:



We would need to iterate for every possible sub-tree that has the left-leaf containing the value 2 and the value 3 which is not efficient.
So in general we are not going to use factorization-trees for cyclic conjunctive queries, but we will be doing that for acyclic conjunctive queries.

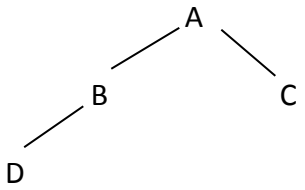
We are going to propose two representations for the materialised view: one extending the factorization-tree based approach for acyclic conjunctive queries and another one inspired by Information Retrieval's Inverted Index for cyclic conjunctive queries.

Question 1 (b) – ACYCLIC CONJUNCTIVE QUERIES

Introduction

Extending factorized-representation

As we said before we will be extending our work on factorization-trees for binary joins.
Our first challenge is given a conjunctive query which layout should a factorization-tree of its result have? After all for n vertices there is a possible number of n^{n-2} labelled trees [4], so which factorization tree to choose?
The conjunctive query R1(A,B), R2(A, C), R3(B,D) has a good factorization-tree in:



From the example we can infer some rules that a good factorization-tree should possess:

- Internal nodes are joining attributes and leaf nodes are non-joining attributes.
- If an internal node has a parent, then the parent is involved in an equal or higher number of relations.
- The parent of any node is a joining-attribute and there exists a relation with the two attributes.

Another important thing is the order of joining attributes, for our above example would be **B A**, the usefulness of this ordering of join attributes will be explained later.
We can sketch then an algorithm for coming up with the above factorization-tree and an order of joining attributes.

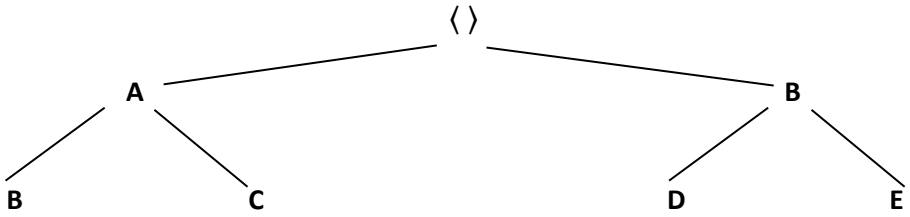
Algorithm for obtaining the factorization-tree from a conjunctive query and the ordering of joining attributes

Function TreeLayout(Query)

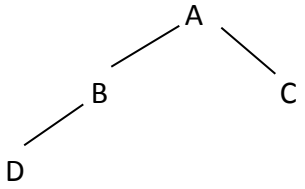
(a) Let *jattributes* be the list of the joining attributes of the conjunctive query
(b) Sort the *jattributes* list by the number of relations in which they are involved in descending order. (ie. in the above example A is involved in 2 relations, B in 1).
(c) Let *f-tree* be an empty tree.
(d) Let *penalty* be an empty priority queue where every joining attribute has value 0 at startup.
(e) For each *attribute* in *jattributes*
    a. For each *relation* involving *attribute*
        i. Let *other_attribute* be the other attribute in the *relation*
        ii. If *other_attribute* is not a joining attribute
            1. Attach *other_attribute* with parent *attribute* in the *f-tree*
        iii. Otherwise:
            1. If *other_attribute* has not already been attached in the *f-tree*
                a. Attach *other_attribute* with parent *attribute* in the *f-tree*
                b. Increment the priority of *other_attribute* in *penalty* by the priority of *attribute* plus *1*
(d) Sort *jattributes* sorted by *penalty* in descending order
(e) Return (*f-tree*, *jattributes*)

[4] *Cayley, A. (1889). "A theorem on trees". Quart. J. Math **23**: 376–378.*

The factorization-tree that we would obtain with the above algorithm for the conjunctive query R1(A,B),R2(A,C),R3(B,D),R4(B,E) would be the following (⟨ ⟩ represents an empty relation, it is put just for clarity purposes):



Thorough the section dedicated to the handling acyclic binary queries we will be using as example this factorization-tree and the conjunctive query R1(A,B), R2(A, C), R3(B,D) :

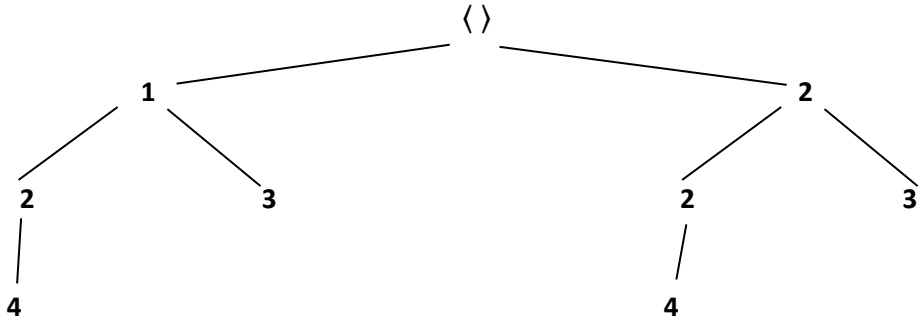


We extend the original factorization-tree described in previous work with a new feature: *compaction of sub-trees*.

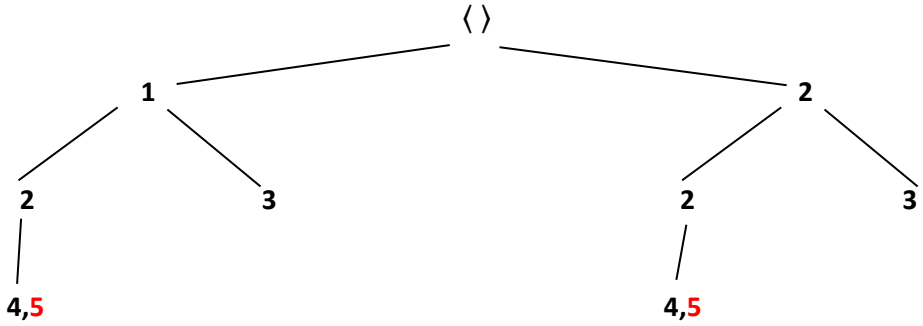
Let's imagine we wish to materialize the following result of the query:

A	B	C	D
1	2	3	4
2	2	3	4

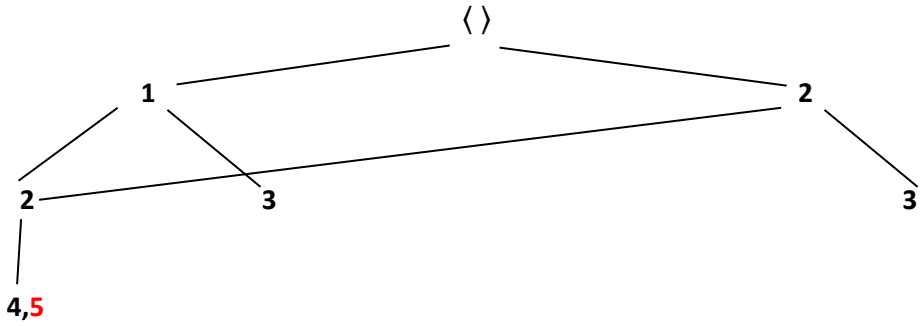
The factorization-tree is then given by:



Let's imagine we want to insert the record $(2,5)_{R_3}$, then the updated factorization-tree is:



Remember that we are talking about acyclic-joins so every parent joining with another joining attribute inherits all the sub-results involving all its child other attributes, to put more simply in this case makes more sense to compact the sub-tree containing the attribute B.
So instead for the above example we will have:



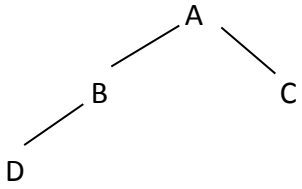
But in general we compact every sub-tree for every joining attribute to avoid useless repetition.

The idea is that every subtree's parent has two type of child:

- Leaf child, they represent non joining-attributes and thus are represented with a set.
- Non-leaf child, they represent joining attributes and thus they are represented with a pointer to another sub-tree.

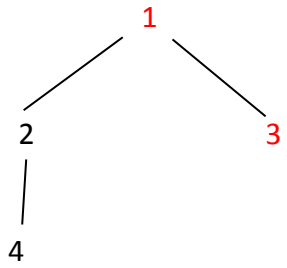
Under inserts

The first step is given a result of the query to build the resulting factorization-tree, in this step the order of the joining attributes previously described will be important for estabilishing an invariant property: if an attribute in the layout of the factorization-tree has among its childs another joining attribute then we must proceed first in creating the first sub-tree.
Example, given the factorization-tree and the conjunctive query R1(A,B), R2(A, C), R3(B,D):



And the row of the result **A**=1 **B**=2 **C**=3 **D**=4, we first proceed in creating the sub-tree with attribute **B**:

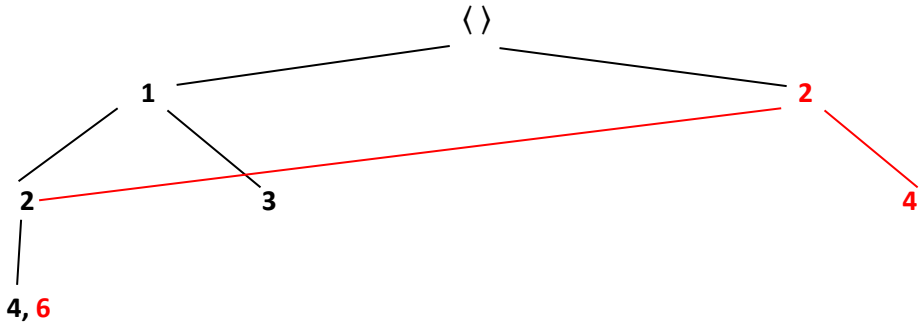
And then in a second stage we proceed in creating the sub-tree with attribute **A** and attach the previously created sub-tree:



So the idea is to create a single sub-tree at time by starting from the bottom to the top, this is why we need an order of the joining attributes which we can obtain with the algorithm described before.

Suppose now that we want to update the it with the result: **A=**2 **B**=2 **C=**1 **D**=6:

- We first proceed in creating the sub-tree of the **B** and **D** attributes but we can note that the appropriate sub-tree already exists so we can retrieve that and check if we need to update it with the value of the other non-joining attribute (**D**).
- We create the sub-tree of the **A** and **D** attributes.



Algorithm for building/updating the factorization-tree from a query result

Function BuildFactorizationTree(Result, f-tree, FactorizationTreeLayout, OrderJoinAttributes)

(a) For Each *Row* in the Result
  a. For *attribute* in OrderJoinAttributes
    i. Let *t* be the value of the joining attribute *attribute*
    ii. If there exists a sub-tree with root *t* in *f-tree*
      1. Create a sub-tree with root *t*
    iii. Otherwise just retrieve the existing sub-tree
    iv. For Each *sub_attribute* in the child of *attribute*
      1. Let *field_value* be the value of *sub_attribute* in *Row*
      2. If *sub_attribute* is a joining attribute
        a. Attach the sub-tree with root *field_value* to the created sub-tree
      3. Otherwise
        a. Insert *field_value* in the correct leaf of the sub-tree

The complexity of the algorithm is: $O(|R| * |A|)$ where R is the result and A the attributes of the result.

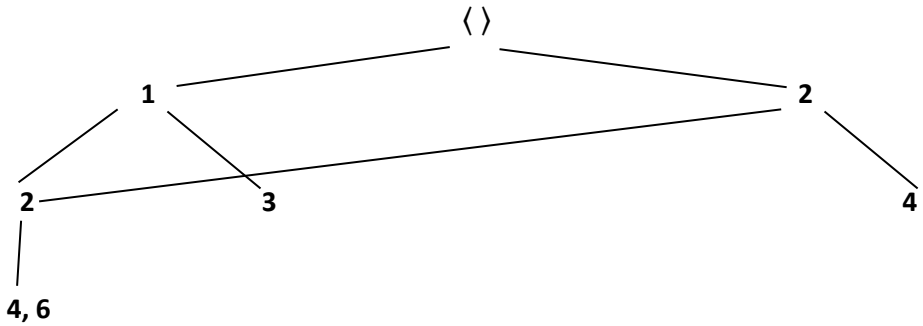
Now we can proceed in giving the algorithm to update the Factorization-tree after an insert.

We remember that we assume that the tables are binary and we differentiate them in two categories:

- Tables where both the attributes are joining-attributes
- Tables where just one of the attributes is a joining-attribute

When we insert a new Record we are interested in obtaining the most dominant joining attribute as it will tell in which sub-tree we need to operate.

Suppose we have the below factorization-tree of the result and we are adding the record $(3, 10)_{R_1}$:



The record $(3, 2)_{R_1}$'s most dominant joining-attribute is **A** so we will before check if there exists a proper sub-tree which is not present in the above case so we proceed in running a light-weight join using the same trick as before: $(3, 2)_{R_1} \bowtie R_2 \bowtie R_3$ and then pass the result to the algorithm to update the factorization-tree described before which will recognize that a sub-tree with label **B=**2 already exists so it will take care of linking that to the newly created sub-tree for the label **A=**3.

In the case both the attributes of the inserted record are joining attributes and the sub-trees of each attribute do not exist in the factorization-tree we still need to run a join query.

The approach described in above is very similar in nature to what we proposed for binary-joins, the difference is that we check whether the record has only 1 joining-attribute or both, as sometimes we do not need to run a join query to obtain the entries to add to the result, we run a join query in the worst-case scenario and the query is even smaller to the full-fledged join that NaiveDB runs for computing the result of the query from scratch.

Algorithm for updating the factorization-tree upon insertion of a new record

Function UpdateOnInsert(Factorization-Tree, Relation, Record):

(a) Let *attribute* be the joining attribute of *Record.*
(b) If both the attributes of *Record* are joining attributes pick the one that has the higher height in the layout of the factorization-tree (you can also use the order of the joining attributes and pick the right-most one)
(c) Let *other_attribute* be the other attribute of *Record*
(d) Let *t* be the value of *attribute* in *Record*
(e) Let *v* be the value of *other_attribute* in *Record*
(f) If there exists a sub-tree with value *t*
    a. If *other_attribute* is a joining attribute
        i. Check if there exists a sub-tree with value *v*

ii. If there is just attach it to the sub-tree of *attribute* and terminate here

b. Otherwise

i. Just attach *v* to the correct leaf of the sub-tree of *attribute*

ii. Terminate here

(g) Run the join with the Record and all the relations except the one in which we inserted *Record*

(h) Let *Result* be the result of the Join

(i) Then just build the factorization-tree using the previous algorithm *BuildFactorizationTree*

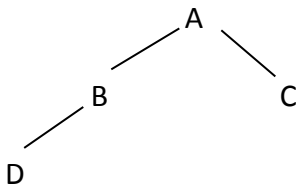
The complexity of the algorithm is $O(|D| + |R| * |A|)$:

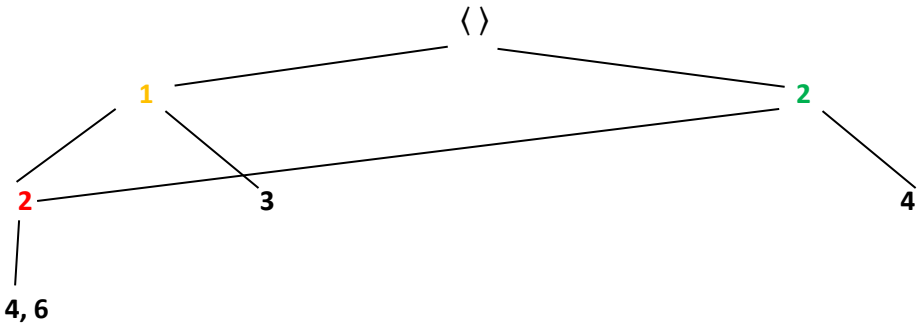
- Hash-join takes complexity linear in size of the relations involved in the join, D represents all the other relations in the database except the one into which we are adding the record.
- Updating the factorization-tree has cost linear in the size of the result and of the attributes involved in the result.

Under deletions

Suppose the given conjunctive query

R1(A,B), R2(A, C), R3(B,D):



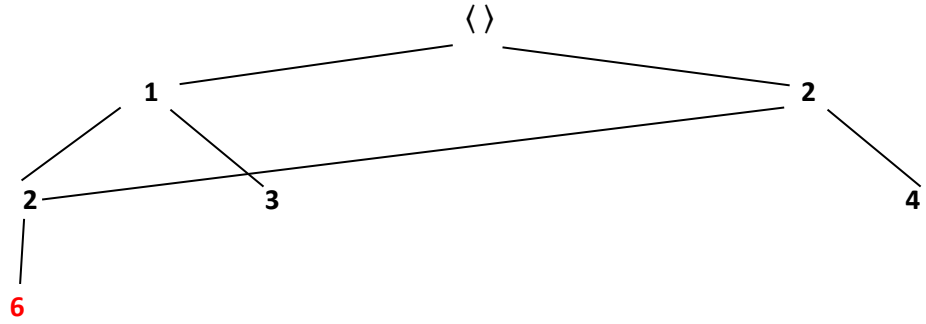


In the above example factorization-tree the sub-tree for value **B**=2 has two parents:

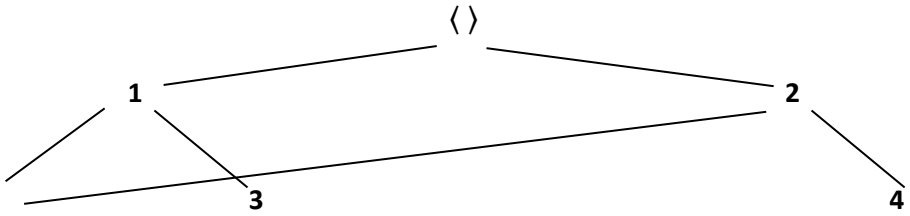
- The sub-tree of **A**=1
- The sub-tree of **A**=2

Given a record deleted from one relation you could say that the deleting operation in the factorization-tree is easy, check if the deleted record exists in the factorization-tree and operate the same way we did for binary joins, well it turns that it is not that easy and we will explain why.

Imagine that from the above example factorization-tree we proceed to delete the record $(2,4)_{R_3}$



And then we delete the record $(2,6)_{R_3}$:

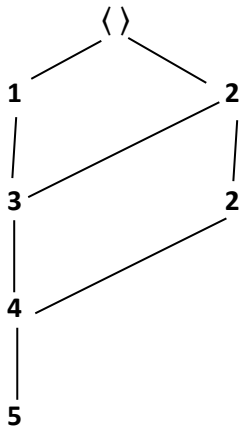


The precedent deletion effectively deleted a sub-tree in the factorization-tree because we emptied the leaf containing the value 6 and in turn we also deleted the parent, it turns out that we must propagate this operation to the parents of the deleted sub-tree and to any child representing join attributes (if any).
In a nutshell if we propagate the deletion of the sub-tree to the parents **A**=1 **B**=2, the parents will get aware that one of the leafs got emptied so we need to delete them as well, thus at the end we obtain the empty result relation:

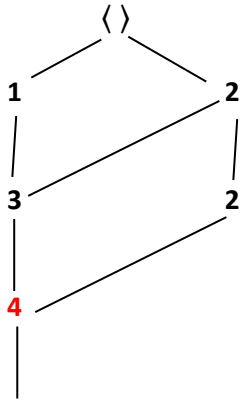
⟨ ⟩

Let's now give another example, the conjunctive query R1(A,B), R2(B,C), R3(B,D)

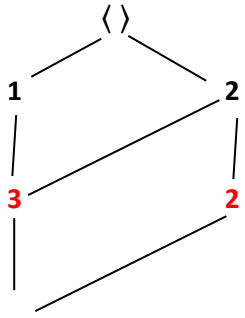
And the result of the above conjunctive query being:



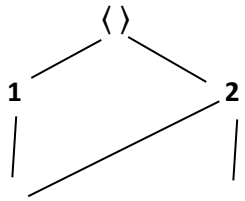
Suppose that we are deleting $(4,5)_{R_3}$, the first step is to check whether a sub-tree for **C**=4 exists and empty the non-joining attribute's value from the leaf:



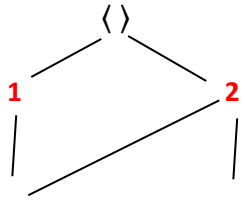
We emptied a leaf of the sub-tree holding **C**=4 so:



We now need to propagate the deletion upward:



Whoa! We have deleted other two sub-trees?! So we need to propagate again!



And at the end we obtain the following:

⟨ ⟩

In a nutshell we have to propagate deletion of a sub-tree both upward and downward:

- Upward we need to check that for a given attribute the parent still has some other value, which means that the leaf of the parent for that attribute is not empty otherwise we need to put the parent in the blacklist (list of the sub-trees to delete).
- Downward, we differentiate between child joining attributes and child non-joining attributes:
    - For a child joining-attribute we just remove the parent's pointer from the child sub-tree, then we check if removing the pointer from the list has emptied the child sub-tree list of parent's pointers in which case we proceed to put that in the blacklist.

The idea is that we proceed to delete the record from the factorization-tree if present in the result and then we check if we emptied a leaf of the corresponding sub-tree.

Algorithm for updating the factorization-tree upon deletion of a record

Function UpdateOnDelete(Factorization-Tree, Relation, Record):

(a) Let *attribute* be the joining attribute of *Record.*
(b) If both the attributes of *Record* are joining attributes pick the one that has the higher height in the layout of the factorization-tree (you can also use the order of the joining attributes and pick the right-most one)
(c) Let *other_attribute* be the other non-joining attribute or the less dominant joining attribute in the factorization-tree.
(d) Let *value* and *other_value* be respectively the *attribute* and *other_attribute* values.
(e) If there exist no sub-tree for *attribute* with value *value* terminate here
(f) Let *t* be the sub-tree for *attribute* with value *value*
(g) If *other_attribute* is not a joining-attribute
    a. Delete *other_value* from the *other_attribute*'s leaf of *t*
(h) Otherwise:
    a. Let *w* be the sub-tree for *other_attribute* with value *other_value*
    b. Delete from the *other_attribute* leaf of *t* the pointer to the sub-tree *w*
(i) If *t* has some empty leaf:

a. Let *black_list* be an empty list
b. Let *toVisit* be a queue
c. Put *t* sub-tree in *toVisit*
d. While *toVisit* non empty
    i. Let *z* be a sub-tree taken from *toVisit*
    ii. Start inspecting upward nodes
        1. For every parent of *z*
            a. Delete *z* from the parent sub-tree's leaf
            b. If we emptied a leaf of the parent put the parent in the *black_list* and in *toVisit*
    iii. Start inspecting downward child nodes of *z*
        1. For every child joining attribute of *z* in the factorization-tree layout
            a. For every child sub-tree of *z*
                i. Remove from the child's parent list of pointers the pointer to *z*
                ii. If we emptied the child's parent list of pointers then put the child in both *black_list* and *toVisit*
e. Delete every sub-tree in the *black_list*

Categorizing the complexity of the algorithm is very complicated because it is linear in the size of the rows of the result to delete but we can still make the case that we outperform NaiveDB due to the fact that NaiveDB runs a join-query after deleting the record and we perform an inferior amount of work, after all we are just removing every link of the deleted sub-trees from the deleted portions of the result.

Data structures

So the idea is that every joining-attribute's value of the factorization-tree has its own sub-tree so what we need is a mapping:

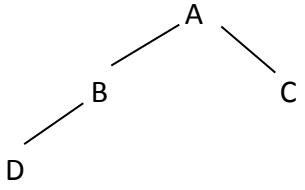
$$joining\_attribute \rightarrow value \rightarrow subtree$$

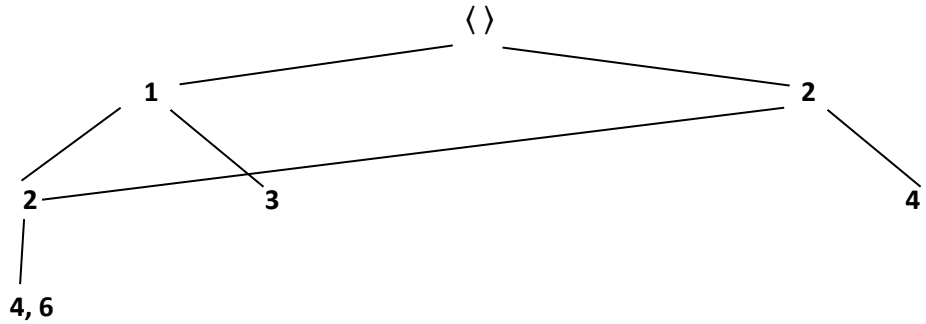
We use a two level hash-table with the first level indexing the joining attributes and the second level the value of that attribute from which we can obtain the corresponding subtree.

The subtree is implemented through a structure that holds the root joining attribute itself and the value of that attribute.
Childs attributes of the subtree's root attribute are represented through a dictionary where:

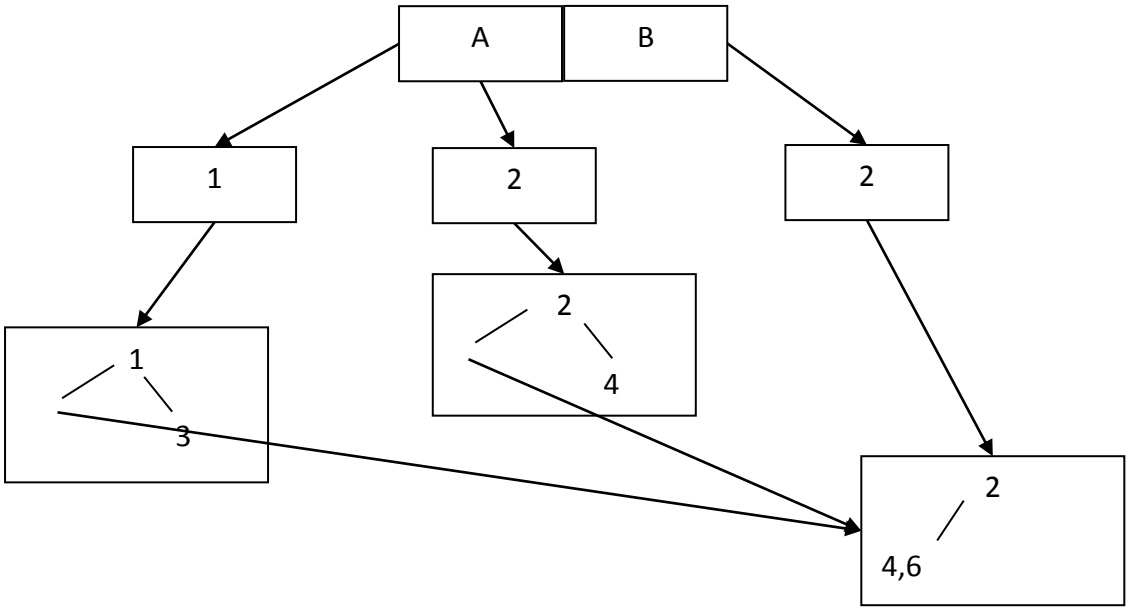
- Every non-joining attribute's child points to a set of values for that attribute
- Every joining attribute's child points to a list of pointers to other sub-tree

Let's take for example the factorization-tree:





The above factorization-tree will be represented in memory as:

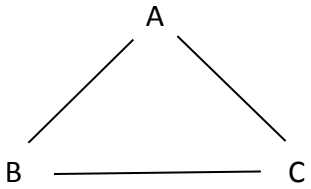


Question 1 (b) – CYCLIC CONJUNCTIVE QUERIES

Introduction

Borrowing some tricks from information retrieval and combining it with factorization-trees

In the case of a cyclic query like R1(A,B), R2(A,C), R3(B,C) we showed that factorization-tree is harder to maintain so we opted for a different representation for this class of cyclic queries

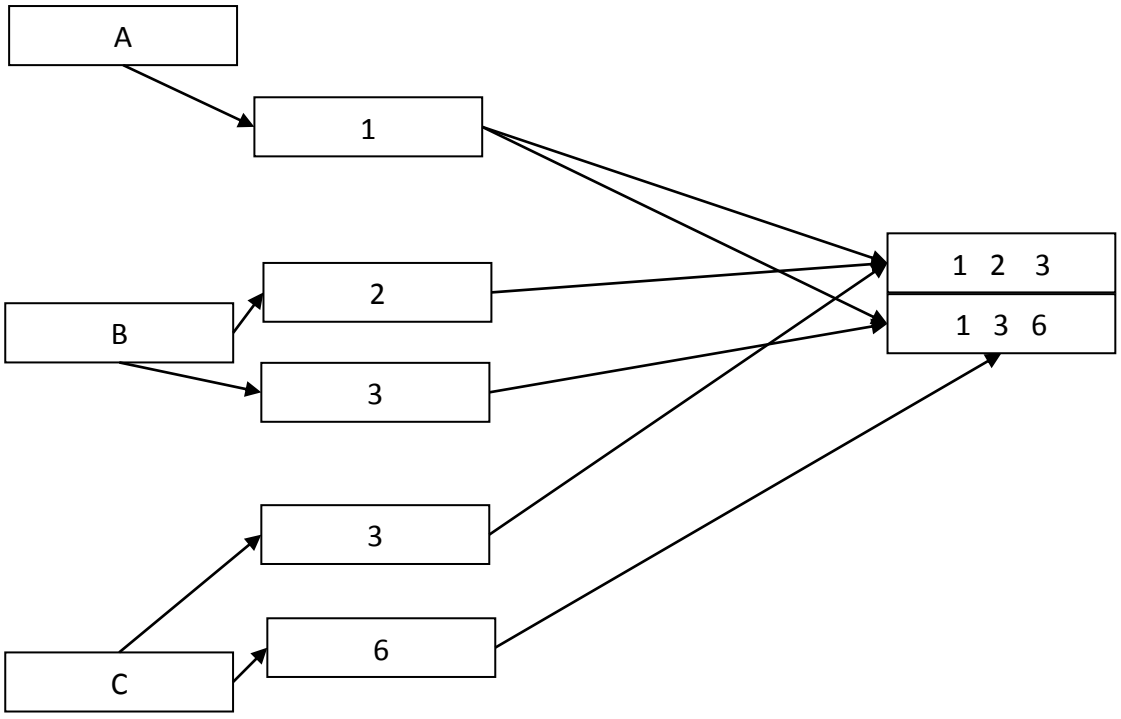


We will use a representation inspired by Information Retrieval's inverted indices.

So for example let's assume that the result of the conjunctive query R1(A,B), R2(A,C), R3(B,C) is:

A	**B**	**C**
1	2	3
1	3	6

The way they will be represented is:



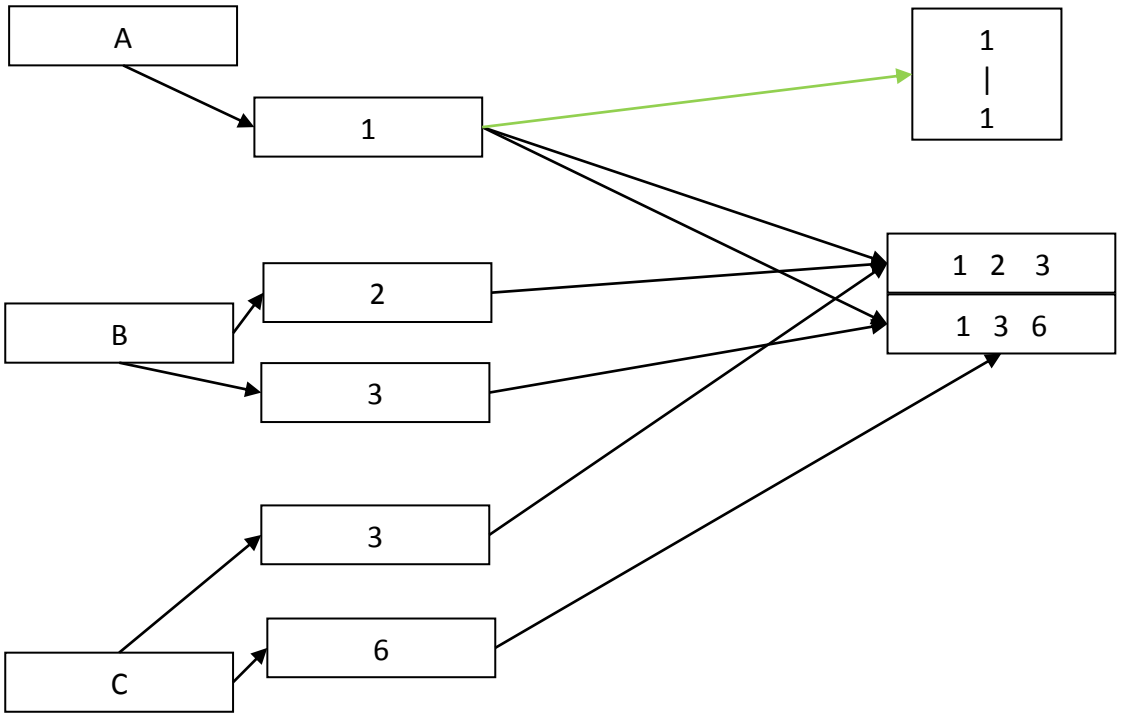
The idea is that every joining attribute will have its corresponding index pointing to a structure holding the result of the query. And for non-joining attributes? For non-joining attributes we exploit the assumption that the tables are binary so every non-joining attribute is paired with a joining-attribute and for them we are going to use factorization-trees

Let's assume the conjunctive query R1(A,B), R2(A,C), R3(B,C), R4(A,D) with the result being:

A	**B**	**C**	**D**
1	2	3	1
1	3	6	1

The idea is that we split the query in a cyclic part containing the joining-attributes and an acyclic part for every joining-attribute that can be paired with another non-joining attribute.
So for the above query we a tabular representation of **A B C** and a Factorization-Tree for the acyclic part of the result **A D** that can be represented with the simple factorization-tree:

The above will be represented in memory as:



Under insertions

We recall the trick of before, let *t* be the added record to a table we can obtain the list of entries to be added to the result by running a light-weight join between *t* and all the other relations excluding the one in which we added the record *t*.

We categorize two different type of insertions, one involving a relation where there is a non-joining attribute and the other where both of the attributes of the relation are joining attributes.

Let's start by talking about insertions to a table where both the attributes are joining attributes in the conjunctive query: in this scenario we need to use the trick and run the light-weight join query with the new added record and all the other relations, this is the worst-case scenario under this representation.

In the case the newly inserted record has a non-joining attribute we can check before whether there exists a factorization-tree sub-tree with the value of the joining attribute, if that is the case we simply attach the non-joining attribute value to the leaf of the sub-tree otherwise we run the light-weight join query.

Now we describe the algorithm to update the hybrid Inverted-Index Factorization-Tree representation from a query result.

Algorithm for building/updating the factorization-tree from a query result

Function BuildInvertedIndex(Result, Inverted-Index, Factorization-Tree)

(a) For Each *Row* in the Result
    a. For each *joining_attribute* in *Row*
        i. Let *v* be the value of the *joining_attribute* of *Row*
        ii. Add the pointer to *Row* in *Inverted-Index[joining_attribute][value]*
    b. For Each *joining_attribute* involved in a relation with another *non-joining attribute*
        i. Let *v* be the value of the *joining_attribute* of *Row*
        ii. Update the Factorization-Tree for *joining_attribute* with *v*

Now it is time to describe the algorithm to update the materialized view upon insertion of a new record.

Algorithm for updating the inverted-index upon insertion of a new record

Function UpdateOnInsert(Inverted-Index, Relation, Record):

(a) If the Relation has non-joining attributes
    a. Let *joining_attribute* be the joining attribute of the relation
        i. Let *v* be the value of the *joining_attribute* of Record
        ii. If there exists a factorization-tree for *joining_attribute* with value *v*
            1. Update the factorization-tree by attaching the value of the other attribute of Record
            2. Terminate here
(b) Run the join with the Record and all the relations except the one in which we inserted *Record*
(c) Using the previous algorithm BuildInvertedIndex update the materialized view

In general it is always bad when we add a new record in a relation where both the attributes are joining attributes as we are always forced to execute a join query in order to update the materialized view, this is our worst-case scenario.

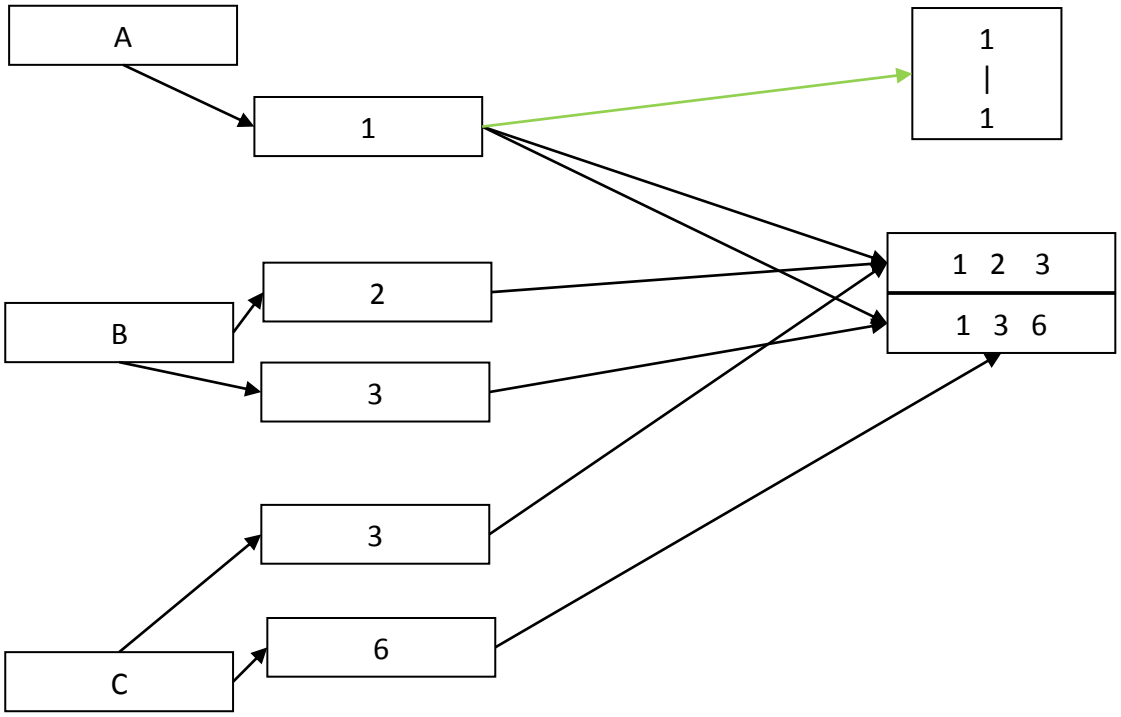
Let's give an example, suppose the conjunctive query R1(A,B), R2(A,C), R3(B,C), R4(A,D) and input tables:

R1		**R2**		**R3**		**R4**	
A	**B**	**A**	**C**	**B**	**C**	**A**	**D**
1	2	1	3	2	3	1	1
1	3	1	6	3	6		

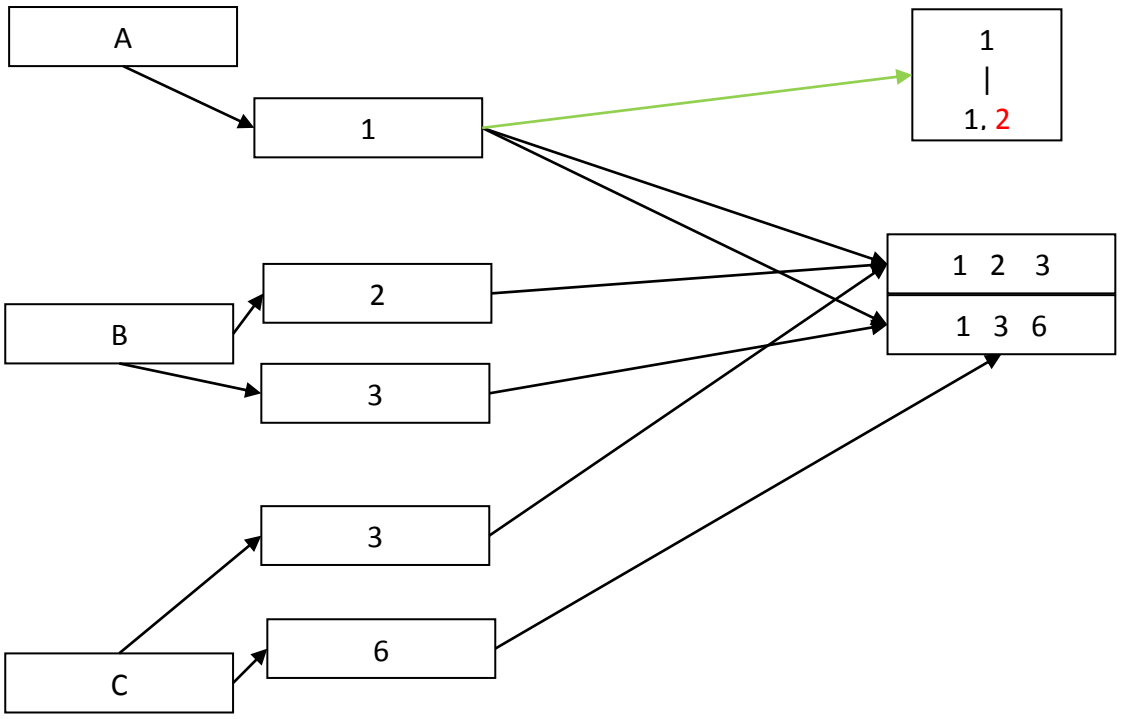
The result of the conjunctive query over the relations above is:

A	**B**	**C**	**D**
1	2	3	1
1	3	6	1

Which is represented through this inverted index structure:



Suppose we are adding the tuple $(1, 2)_{R_4}$, we first realize that the relation has just one joining attribute so we check whether there exists a sub-tree for the joining attribute of the relation (***A***) with value ***1***, which in the above example exists so we can simply update the sub-tree:



Now let's imagine that we are adding the tuple $(2, 10)_{R_2}$, this is bad because all the attribute of *R2* are joining attributes so we are forced to run the join query $(2, 10)_{R_2} \bowtie R_1 \bowtie R_3 \bowtie R_4$ in order to check whether there are new entries to add to the above data structure.

This outperforms NaiveDB because the workload of the join query to run in the worst-case scenario is smaller compared to run a join query on all of the relations of the database. Updating the data structure is linear in size of the result and in number of the attributes.

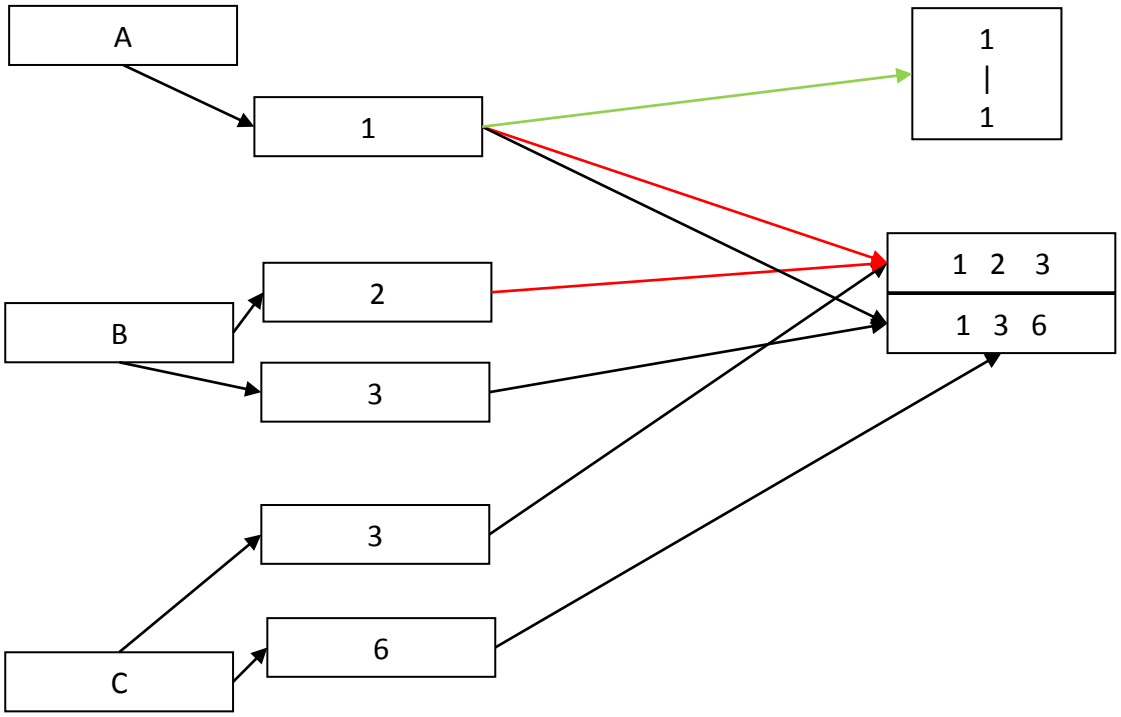
Under deletions

In general our hybrid approach of factorization-tree and inverted indexes make deletions easy to manage, we differentiate two types of deletions: deletions involving tuples with both the attributes being joining attributes or tuples with just a single joining attribute.

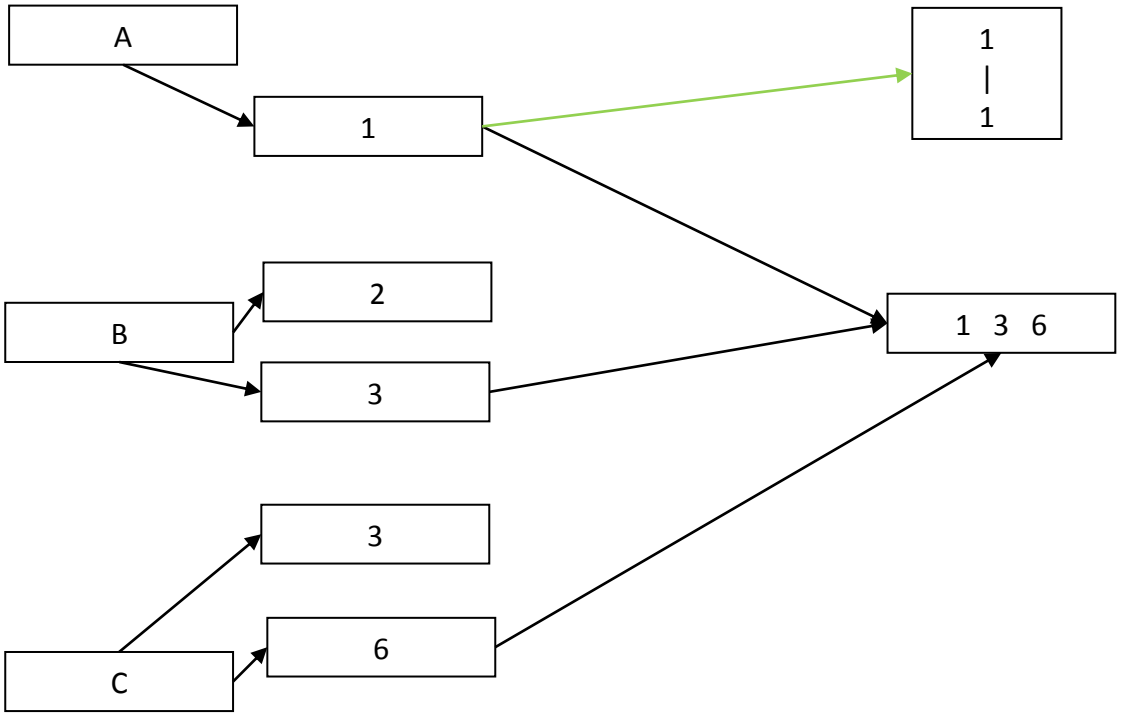
Let's give an example, suppose the conjunctive query R1(A,B), R2(A,C), R3(B,C), R4(A,D) and input tables:

R1		**R2**		**R3**		**R4**	
A	**B**	**A**	**C**	**B**	**C**	**A**	**D**
1	2	1	3	2	3	1	1
1	3	1	6	3	6		

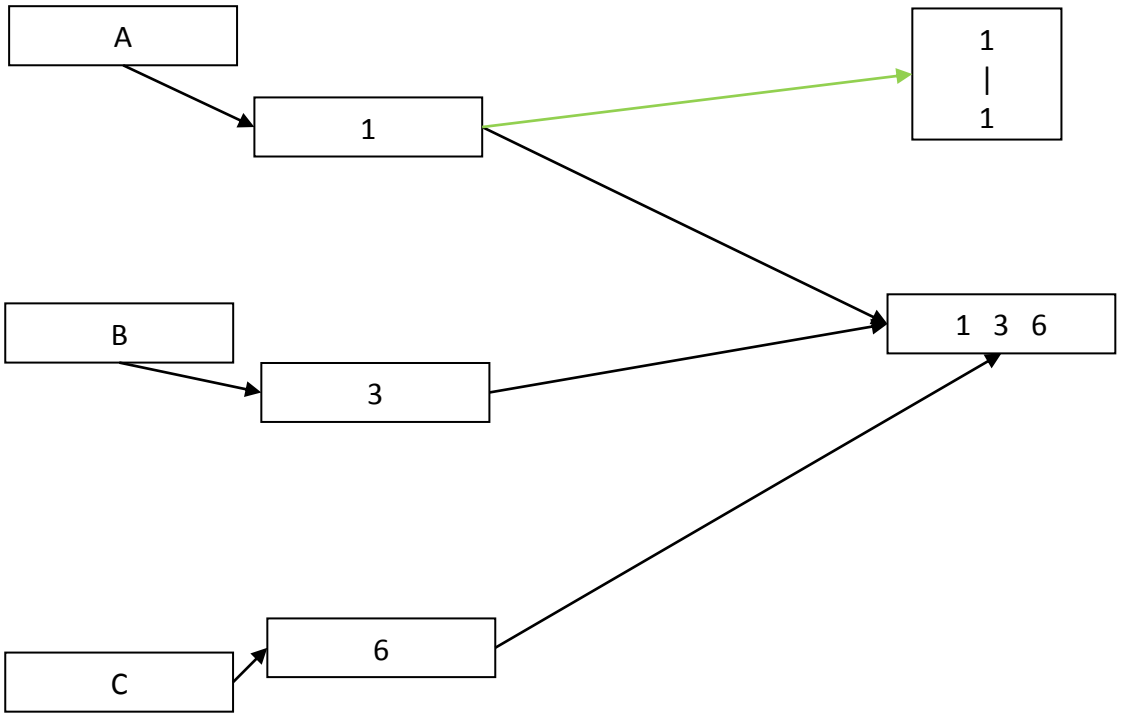
Suppose we are deleting the tuple $(1,3)_{R_1}$.
What we do is to intersect the results of the index **A**=1 and **B**=3.



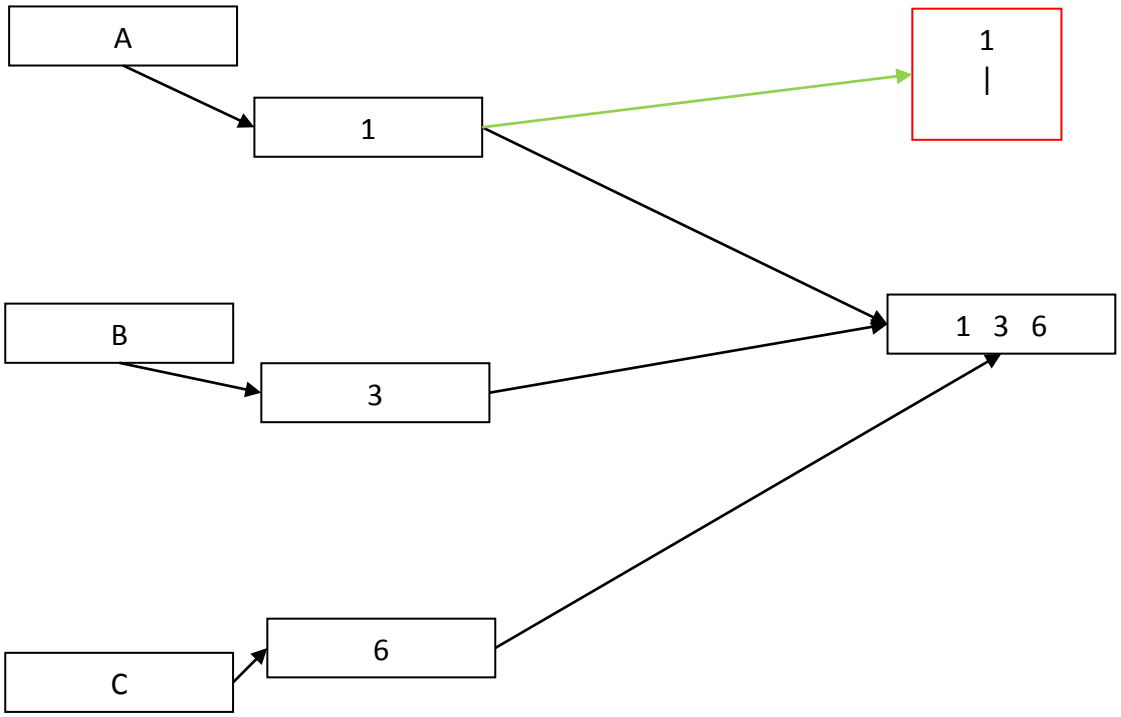
In red are the links that intersect in both the index.
Then for each result contained in the intersectio we shall remove them from the inverted index and check whether if we have emptied some buckets.



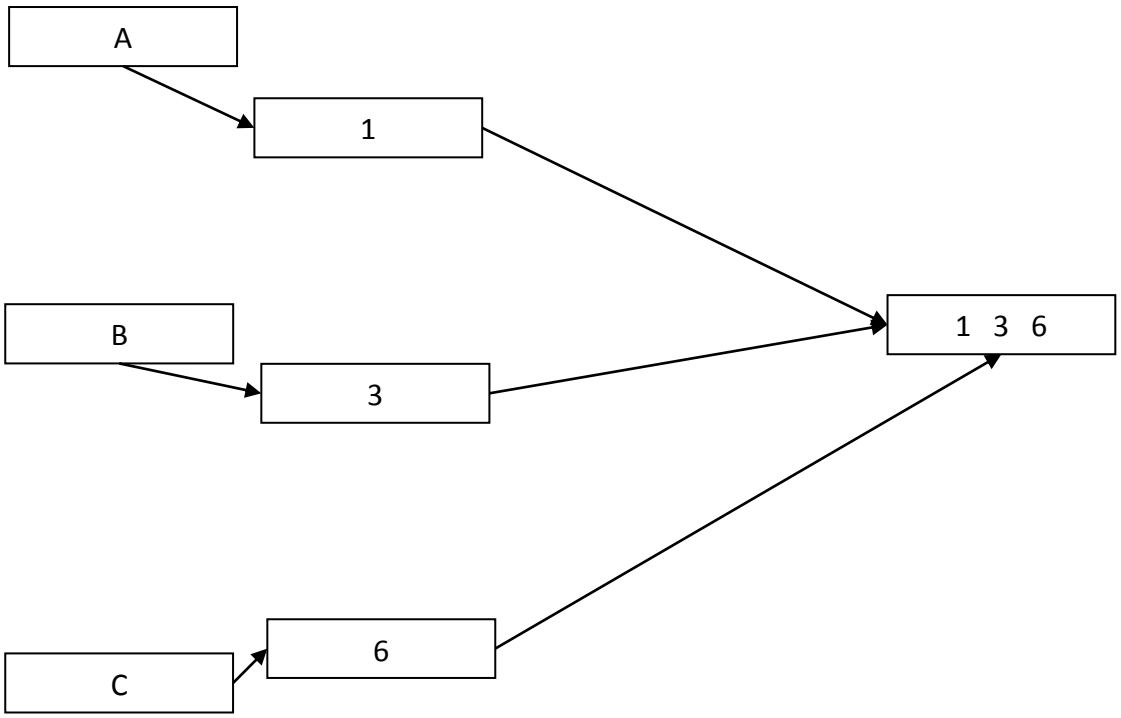
As you can see we have emptied the buckets **B**=2 and **C**=3 so we shall do some clean up on the inverted index.



Let's imagine that now we proceed in deleting $(1,1)_{R_4}$.
R4 is a relation with a single joining attribute (**A**) so we check whether exists a sub-tree with value **1** and then proceed to remove the non joining-attribute value from the leaf of the sub-tree:

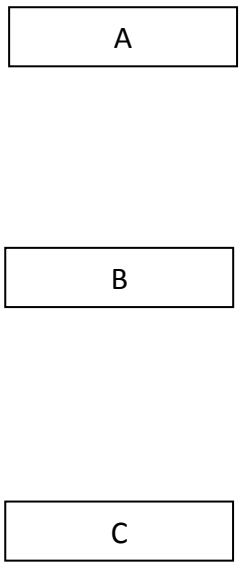


We have emptied all the leaf of the sub-tree so we shall proceed in deleting that.



But now **A=**1 has no sub-tree associated with that meaning that we have no result for the factorization-tree, thus there is no row in the result with **A**=1.

So we shall proceed in in deleting every link for **A**=1 which causes in our case to empty the materialized view:



The following is the procedure for removing results in the cyclic art

Algorithm for removing the deleted tuples from the above data structure
(*DeletedTuples* is a list of the tuples from the cyclic part to delete, *PurgeFactTree* is a flag for removing the factorization-tree for acyclic joins when we empty a bucket from the InvertedIndex**)**

Function Purge(Inverted-Index, DeletedTuples, PurgeFactTree):

(a) For Each *Tuple* in *DeletedTuples*
    a. For each *joining_attribute* in *tuple*
        (a) Let *v* be the value of the *joining_attribute* of *Tuple*
        (b) Remove the *tuple* from the bucket (*joining_attribute, v*)
        (c) If the bucket (*joining_attribute, v*) has been emptied
            1. If PurgeFactTree is True Remove the sub-tree (*joining_attribute, v*) if *joining_attribute* is involved in an acyclic relation.
    b. Delete *Tuple*

Complexity of the above algorithm is linear in the number of the result to delete (R) and in the number of the *joining attributes* (A)of the materialized view which yields complexity $O(|R| * |A|)$ We can then proceed in define the procedure for updating the data structure upon deletion of a record.

Algorithm for updating the factorization-tree upon deletion of a record
Function OnDelete(Inverted-Index, Relation, Record):

(a) If *Relation* contains a non-joining attribute
    i. Let *v* be the value of the *joining_attribute* of *Record*
    ii. Let *v'* be the value of the other non-joining attribute of *Record*
    iii. Remove *v'* from the sub-tree (*joining_attribute, v*)
    iv. If the sub-tree (*joining_attribute, v*) has any empty leaf
        1. Remove the sub-tree (*joining_attribute, v*)
        2. Let *deletedTuples* be all the tuples in the bucket (*joining_attribute, v*) of the inverted index
        3. *Purge(Inverted-Index, deletedTuples, False)*
(b) Otherwise:
    i. Let (*v*, *v'*) be the pair of the *joining attributes* of *Record*
    ii. Let *deletedRecords* be the intersection of the *joining attributes* buckets in the inverted index with value *(v, v')*
    iii. *Purge(Inverted-Index, deletedTuple, True)*

A study on the complexity of the above algorithm is hard so we focus on the worst-case scenario which is when we delete a record from a relation where both the attributes are *joining attributes*. The cost of the intersection of two buckets B_1, B_2 is $O(\min(|B_1|, |B_2|))$, we also know that the maximum size of the intersection is given by the bucket with the lowest number of entries so plugging all these information we obtain the complexity:

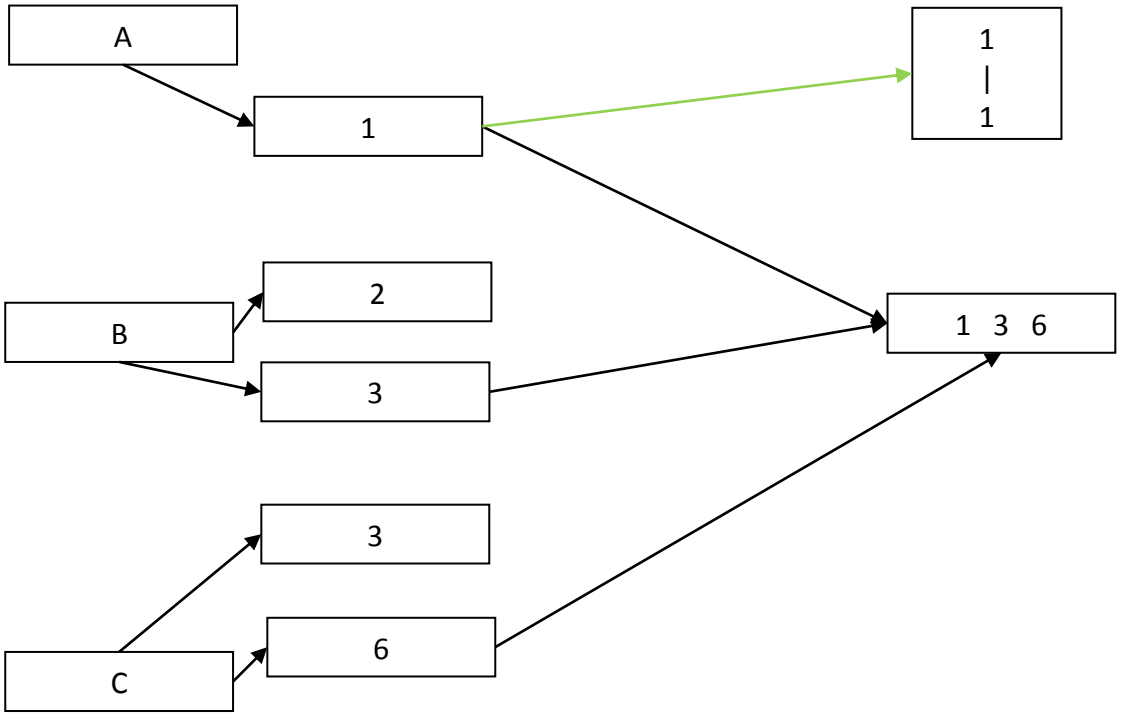
$$O(\ costOfIntersection + costOfPurging)$$

$$O(\ \min(|B_1|, |B_2|) + \min(|B_1|, |B_2|) * |A|)$$

So we can say that it is roughly linear in the size of the smallest bucket, which in the worst-case can contain all the rows of both the relations but we would still be outperforming *NaiveDB* as computing the result of a join has complexity:

$$O(\ |R_1| * \log(|R_1|) + |R_2| * \log(|R_2|) + |R_1 * R_2|)$$

Data Structures



We have seen that the materialized view splits the result in two parts:

- A cyclic part where all the joining attributes are contained
- An acyclic part for every joining attribute involved in relations where there is another attribute which is a non-joining attribute.

The cyclic part portion of the result are represented through a set which at its heart is represented through a hashtable.

The rows of the result are then indexed in all the joining attributes through a 2-level hashtable:

$$attribute \rightarrow value \rightarrow pointers\ to\ records\ in\ the\ result$$

In our data structure we also take core of relations that contains just a single joining attribute which will be stored with factorization-trees specialized for join on a single attribute.
For this regard we extended the solution for binary join of the first question to handle arbitrary joins on a single attribute.

Then we have another hashtable for indexing the subtrees holding the acyclic part of the conjunctive query:

$$attribute \rightarrow value \rightarrow subtree$$

Each record in a relation is still represented as a named tuple which allow retrieval of values from attributes and the relation is represented through a set holding the named tuples.

Question 1 (c)

This question revolves around count query for arbitrary join queries.

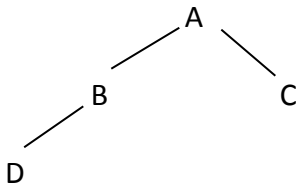
ASSUMPTION
From the question it was not clear whether we were allowed to store count results so we explain how to do counting of results over some representation.

For handling count queries we extend the work done for handling arbitrary join queries by providing algorithms for executing counting over the representations proposed before for both cyclic and acyclic joins.
In a nutshell the data structures used and the algorithms for handling inserts and deletes are the same as in the precedent question but we provide algorithms for the extra step of counting the number of results in the conjunctive query.

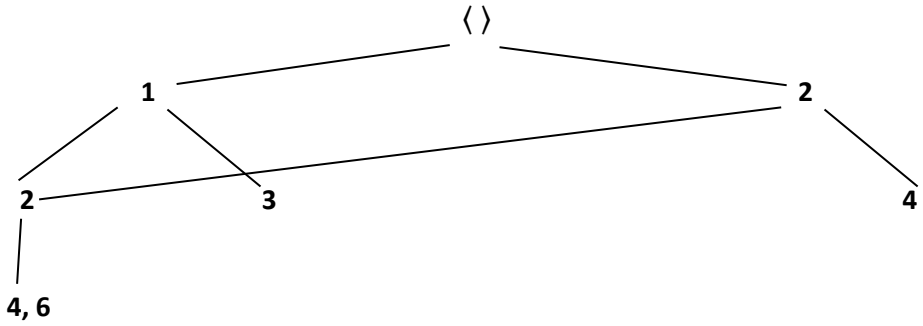
As in question 1.b we proposed two different representations for join queries we provide two algorithms for doing counting over each representation.

Question 1 (c) – CYCLIC JOIN QUERIES

Let's imagine the conjunctive query R1(A,B), R2(A, C), R3(B,D):



Suppose we want to do counting over the following instance of the above factorization-tree:

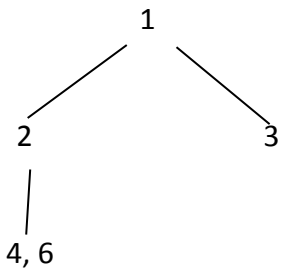


Some notation before:

- $J - CHILD(a)$: this is the set of the child joining attributes of a given attribute a (ie. $J - CHILD(A) = \{B\}$).
- $CHILD(a)$: this is the set of the child non-joining attributes of a given attribute a (ie. $CHILD(A) = \{D\}$
- $CHILD - VALUES(a, v, c)$: this returns the child values with child attribute c of subtree for attribute a, value v (ie. $CHILD - VALUES(A, 1, B) = \{2\}$)
- $ROOT - ATTRIBUTES(f)$: this is the set of root attributes for a given factorization-tree, in the above factorization-tree would return $\{A\}$.
- $ROOT - VALUES(a)$: this is the set of values for a specific root attribute for a given factorization-tree, in the above factorization-tree would return $\{1,\ 2\}$.
-

Now our goal is to define an operation $COUNT(a, v)$ that does counting over a subtree with joining attribute a and value v.

Take for instance the subtree for A with value 1:



What would be the result of $COUNT(A, 1)$? In the example it would be 2 as it represents the results:

A	B	C	D
1	2	4	3
1	2	6	3

$$COUNT(A,1) = \prod_{c \in \{D\}} |CHILD - VALUES(A,1,c)| * \prod_{c \in \{B\}} \left(\prod_{v \in CHILD-VALUES(A,1,B)} COUNT(B,v) \right)$$

The idea is that we multiply the number of values presents in the child attributes of the root of the subtree.
We shall remember that we could have some child attribute that is a joining attribute which means that we have to do counting on that recursively.

We now give a generalized formula for counting over a sub-tree with joining attribute a and value v.:

$$COUNT(a,v) = \prod_{c \in CHILD(a\}} |CHILD - VALUES(a,v,c)| * \prod_{c \in J-CHILD(a)} \left(\prod_{v\prime \in CHILD-VALUES(a,v,c)} COUNT(c,v') \right)$$

The idea is that we before start counting over non-joining attributes and then as the child joining attributes are not leaf of the subtree we need to count that as well.

Now on the top of the $COUNT(a, v)$ procedure we can proceed in giving a generalized formula for an instance of the factorization-tree f:

$$\#RESULT_f = \prod_{c \in ROOT-ATTRIBUTES(f)} \left(\sum_{v \in ROOT-VALUES(c)} COUNT(c,v) \right)$$

Why is this better than the NaiveDB's count query that iterates over each row of the result? The argument is the same as in the question 1.a regarding count over factorization-tree for binary joins.
As the join is acyclic the maximum number of covering relation is 1 meaning that the upper bound of the factorization-tree for the conjunctive query over a database instance D is $O(|D|)$ which means that in the worst case scenario the count is linear in the number of results of the join but as we the result entries get compressed more and more we obtain the count with a fewer number of iterations as we benefit of the compressed representation.

Question 1 (c) – ACYCLIC JOIN QUERIES

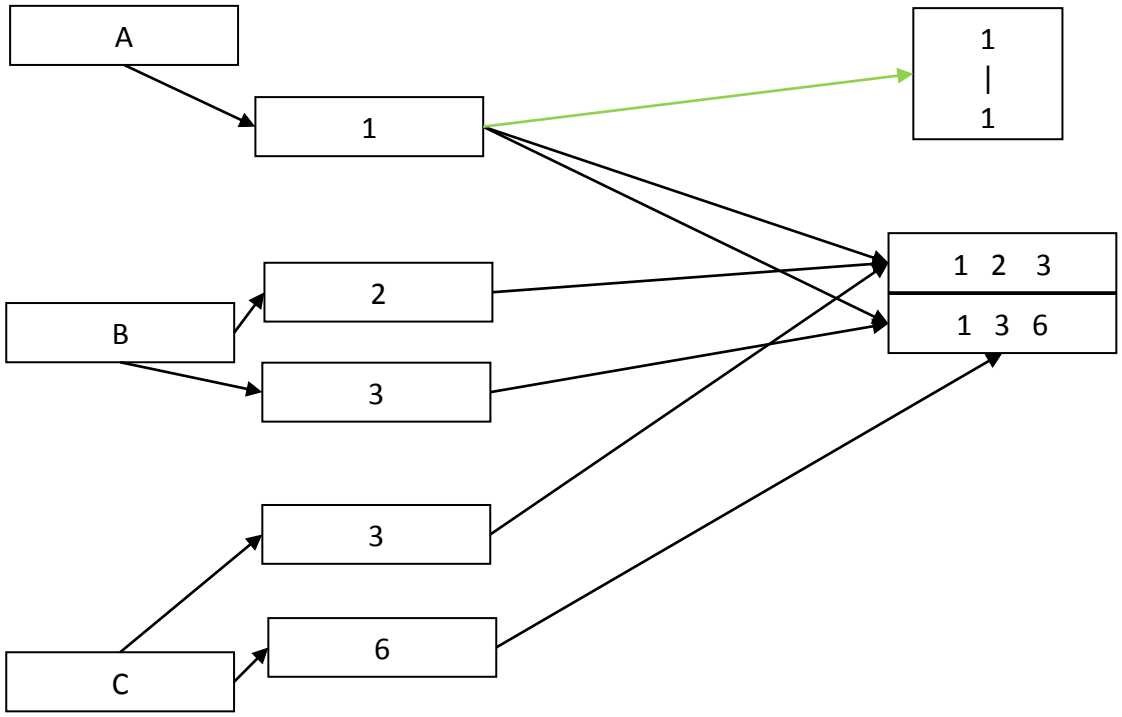
Let's give an example, suppose the conjunctive query R1(A,B), R2(A,C), R3(B,C), R4(A,D) and input tables:

R1		**R2**		**R3**		**R4**	
A	**B**	**A**	**C**	**B**	**C**	**A**	**D**
1	2	1	3	2	3	1	1
1	3	1	6	3	6		

The result of the conjunctive query over the relations above is:

A	**B**	**C**	**D**
1	2	3	1
1	3	6	1

Which is represented through this inverted index structure:



The above representation splits the result in two parts an acyclic part represented through factorization-trees and a cyclic part represented through a set (which at its heart is represented through a hashtable).

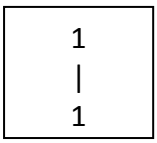
We start memorizing the count for every acyclic entry by storing the result in a temporary 2-level hash-table:

$$attribute \rightarrow value \rightarrow count$$

In the given example we would memoize:

(A, 1) = 1

As the subtree:



Contributes for a single result entry.
We have already covered how to do counting over factorization-tree so we assume that is clear to the reader (see my answer to question 1.a for more details).

After we have memoized the result we would need to iterate over each row of the cyclic part:

1 2 3
1 3 6

So on our first iteration we have

1 2 3

Then for every joining attribute that is paired with other non-joining attributes (in the above example **A**) we should fetch the value for that attribute and obtain the number of acyclic join results for the given attribute and value and we should multiply them, the result of the multiplication is the number of results that have that cyclic part as prefix.
So for the above example we have that **A** is paired with **D** and has value **1**, so we look in the memoized table for entry (**A**, **1**) and we obtain 1, so 1 is the total number of entries with prefix "**1 2 3**").

And then we repeat the process for every iteration.

So now the question is: why do we memoize the number of results for the acyclic part?
The response is: in order to avoid to recompute the count of each subtree over and over, we do that just once and then we memorize that.

Is this better than NaiveDB? Not always, as if we have a fully cyclic join (meaning that we have no acyclic part) then we would be simply iterating over each row, so in that scenario we have no particular gains.

In the scenario a conjunctive query has an acyclic portion we have a gain as we compute the partial count for the acyclic part once and then we proceed in iterating the cyclic part and explore in how many different ways every entry can be combined as a prefix in the result.

Question 2

For each experiment I will be attaching only the code strictly pertinent to the question (all the rest is attached in another paper by the title “All the rest”).

Just a reminder as count queries extend in our proposal extend the work done for handling normal conjunctive queries we just created a `count()` method for counting in each proposed representation implementation (meant to be called after the respective `onInsert()` and `onDelete()` method for updating the materialized view subsequent every insert/delete query) so the code for question 2b and 2c will be grouped.

Question 2 (a)

```
#Code for question 2 (a)
#also used in question 2 (b) for the acyclic part of the Representation for Cyclic joins

from IncDB import IncDB
from join import *
from functools import reduce
from operator import add


#Factored representation for a join on a single attribute (not necessarily binary)
class FactorizedRepresentation_BinaryJoin(IncDB):

    def __init__(self, database, jrels, jattrs = None):
        IncDB.__init__(self, database, jrels)

        #If the joining attribute is supplied then we use that
        if jattrs != None:
            self.jattrs = jattrs
            #In case a joining attribute is supplied it could be the case that the number of
the relations
            #to be joined is 1 in which case the joining attribute would be put in the list
of the normal attributes
            if jattrs in self.otherAttrs:
                self.otherAttrs = tuple(attr for attr in self.otherAttrs if attr != jattrs)
        #Otherwise check if the supplied relations have a joining attribute
        else:
            if len(self.jattrs) != 1:
               raise Exception('This class is meant for factored representation of joins
involving a single attribute')

        #Given that the class is optimized for single attribute joins
        self.jattrs = self.jattrs[0]
        #THE MATERIALIZED VIEW!
        self.view = dict()

        #A list of non-joining attributes
        fields = self.otherAttrs

        #Class for representing a subtree
        class FR_TreeNode(dict):

            def __init__(self):
                for field in fields:
                    self[field] = set()
```

```
        self.Fnode = FR_TreeNode

    #Given a list of tuples representing a new result updates the materialized view
    def fill(self, tuples):
        for tuple in tuples:
            #Obtain the joining attribute's value
            k = getattr(tuple, self.jattrs)
            #If there exists no subtree with the given joining attribute value create it
            if k not in self.view:
                self.view[k] = self.Fnode()
            n = self.view[k]
            #For each non joining attribute...
            for field in self.otherAttrs:
                #...deposit the value in the correct child attribute leaf
                n[field].add( getattr(tuple, field) )

    def onInsert(self, relation, tuple):
        #The insert does not involve a relation involved in the conjunctive query to be
materialized
        if relation not in self.jrels or self.existingRecord(relation, tuple):
            return
        #Value of the joining attribute
        key = getattr(tuple, self.jattrs)
        #Non joining attribute name
        other_attr = [ x for x in tuple._fields if x != self.jattrs ] [0]
        #Value of the joining attribute in the tuple
        value = getattr(tuple, other_attr)
        if key in self.view:
            #If there already exists a subtree deposit the non joining attribute value in
the leaf.
            n = self.view[key]
            n[other_attr].add( value )
        else:
            #Otherwise run a single-record light-weight join query
            #1. Builds the single-record temporary relation
            new_rel = (self.db[relation][0], [tuple], self.db[relation][2])
            rels = [self.db[rel] if rel != relation else new_rel for rel in self.jrels]
            #2. Run the join query
            res = multiway_join(*rels)
            #Check whether there are new results to be put in the materialized view
            if len(res[1]) != 0:
                self.fill(res[1])

    def onDelete(self, relation, rec):
        #The delete does not involve a relation involved in the conjunctive query to be
materialized
        if relation not in self.jrels or self.nonExistingRecord(relation, rec):
            return
        #Joining attribute's value
        key = getattr(rec, self.jattrs)
        #Other non-joining attribute
        other_attr = [ x for x in rec._fields if x != self.jattrs ] [0]
        #Value of the other non-joining attribute
        value = getattr(rec, other_attr)
        #Check if there already exists a sub-tree with the given joining attribute's value
        if key in self.view:
            n = self.view[key]
            #Delete from the corresponding leaf the non-attribute's value
            n[other_attr].discard(value)
            #Any empty child?
            if any(len(n[field]) == 0 for field in n.keys()):
                #If yes then remove the subtree
                self.view.pop(key)
```

```
    def count(self):
        count = 0
        #For every subtree in the current instance of the factorization-tree
        for value in self.view:
            #Subtree
            node = self.view[value]
            #Partial value
            partial = 1
            #The total number of results is given by the product of the number of elements
in each leaf so
            #for each leaf of the subtree...
            for _, child in node.items():
                temp = 0
                #...for every element in the given leaf
                for y in child:
                    #...increase the counter by one
                    temp += 1
                #Update the partial product
                partial = partial * temp
            #Update the global count with the total number of results represented in the
given subtree
            count += partial
        return count

    def result(self):
        return self.view
```

Question 2 (b) (c) – Some notes

As I proposed two different representation for cyclic and acyclic joins, some facility had to be created that given a conjunctive query returns the correct representation to use (it just checks whether there is a cycle in the query graph, for correctness I found online the code for cycle detection in an undirected graph).

```
#Question 2(b), 2(c)
#Facilities for analyzing the conjunctive query
#and determine whether the join is cyclic or acyclic.
#Depending on whether its cyclic or acyclic we return
#the apposite representation.

from FactorizedRepresentation_ArbitraryAcyclicJoins import *
from Representation_ForCyclicJoins import *

#Given a list of relations to be joined
#it returns the representation for the result to be used
#depending on whether the conjunctive query
#is cyclic or acyclic
def obtainRepresentation(relations):
    cycles = find_cycles([rel[0] for rel in relations])
    return FactorizedRepresentation_ArbitaryAcyclicJoins if len(cycles) == 0 else
Representation_ForCyclicJoins

#A graph algorithm to check the presence of cycles found on StackOverflow
#http://stackoverflow.com/questions/12367801/finding-all-cycles-in-undirected-graphs
def find_cycles(graph):
    cycles = []
    for edge in graph:
        for node in edge:
            findNewCycles([node], graph, cycles)
    #paths = [tuple(cy) for cy in cycles for node in cy]
    return [tuple(cycle) for cycle in cycles]

def findNewCycles(path, graph, cycles):
    start_node = path[0]
    next_node= None
    sub = []

    def invert(path):
        return rotate_to_smallest(path[::-1])

    #  rotate cycle path such that it begins with the smallest node
    def rotate_to_smallest(path):
        n = path.index(min(path))
        return path[n:]+path[:n]

    def isNew(path):
        return not path in cycles

    def visited(node, path):
        return node in path

    #visit each edge and each node of each edge
    for edge in graph:
        node1, node2 = edge
        if start_node in edge:
                if node1 == start_node:
                    next_node = node2
                else:
```

```
            next_node = node1
    if not visited(next_node, path):
            # neighbor node not on path yet
            sub = [next_node]
            sub.extend(path)
            # explore extended path
            findNewCycles(sub, graph, cycles);
    elif len(path) > 2  and next_node == path[-1]:
            # cycle found
            p = rotate_to_smallest(path);
            inv = invert(p)
            if isNew(p) and isNew(inv):
                cycles.append(p)
```

Question 2 (b) (c) – ACYCLIC CONJUNCTIVE QUERIES

```
#Question 2(b), 2(c) - Acyclic Joins

from IncDB import IncDB
from join import *
from functools import reduce
from operator import add
from collections import Counter, defaultdict

#Given a list of binary relation attributes and a list of joining attributes
#Returns the layout of the factorization-tree, ie. imagine R1(A,B), R2(A, C), R3(B,D)::
#         A: [B, C]
#         B: [D]
#and a list of weights of the joining attributes from which we can infer the order of the
variables.
def TreeLayout(rel_attributes, jattributes):
    already_processed = set()
    fields = defaultdict(set)
    penalty = Counter()
    #Sort the attributes in descending order by the number of relations in which they are
involved
    occurences = Counter([attr for rel in rel_attributes for attr in rel if attr in
jattributes])
    order_of_attributes = sorted(occurences, key=occurences.get, reverse = True)
    #For each attribute
    for attr in order_of_attributes:
        for rel in rel_attributes:
            if attr not in rel:
                continue
            #For each relation containing the attribute "attr"
            #Obtain the name of the other attribute
            o_attr = rel[0] if rel[0] != attr else rel[1]
            #Check if it has not already been attached in the factorization-tree
            if o_attr not in already_processed:
                #Check if the other attribute is a joining attribute
                #and if it has not already been attached. If it has already been that it
means that
                #the other attribute has attached the current attribute
                if o_attr in jattributes and o_attr not in penalty:
                    #Update the layout of the factorization tree by attacching
                    #other attribute as child of the current attribute
                    fields[attr].add(o_attr)
                    #Penalize the other attribute by penalty of the current attribute plus
one
                    penalty.update({o_attr : penalty[attr] +1})
                    already_processed.add(attr)
                elif o_attr not in jattributes:
                    #otherwise it is not a joining attribute and we can attach it freely!
                    fields[attr].add(o_attr)
    #fields gives us the layout of the factorization tree
    #penalty gives us the order of the joining attributes (they are sorted by the height in
the factorization-tree)
    return fields, penalty


#This represents a subtree
class FactorizedRepresentation_TreeNode:

    def __init__(self, joining_attribute, value):
        self.value = value
        #List containing the parents of this subtree
```

```
        self.parents = set()
        #The joining attribute at the root of this subtree
        self.joining_attribute = joining_attribute
        #The childs of the subtree
        self.childs = dict()

    #Check if the given supplied attribute is a child attribute of this subtree
    def __contains__(self, key):
        return key in self.childs

    #Returns a given attribute leaf
    def __getitem__(self, name):
        return self.childs[name]

    #Updates a child subtree
    def __setitem__(self, name, value):
        self.childs[name] = value

    def keys(self):
        return self.childs.keys()

    #This counts the number of results represented by the subtree
    def count(self):
        count = 1
        #For every child attribute
        for attr, child in self.childs.items():
            partial = 0
            #The child is a non-joining attribute so we just iterate over each
            #element
            if type(child) is set:
                for item in self[attr]:
                    partial += 1
            #Otherwise it is a joining attribute which means every child is a subtree so we
            #just call the apposite count method
            else:
                #For each subtree for the given attribute
                for key, subtree in child.items():
                    partial += subtree.count()
            #Update the partial product
            count *= partial
        return count


class FactorizedRepresentation_ArbitaryAcyclicJoins(IncDB):


    def __init__(self, database, jrels):
        IncDB.__init__(self, database, jrels)
        #Obtain the layout of the factorization-tree and of the
        #order of the joining variables
        self.fields, self.dominance = TreeLayout([database[rel][0] for rel in jrels],
self.jattrs)
        #Initialize the two level hashtable attribute->valuie
        self.index = dict()
        for attr in self.jattrs:
            self.index[attr] = dict()
        #The materialized view
        self.view = list()
        for attr in [ a for a in self.jattrs if self.dominance[a] == 0 ]:
            self.view.append(self.index[attr])

        self.Fnode = FactorizedRepresentation_TreeNode
```

```
    #Given a set of tuples update the factorization-tree
    def fill(self, tuples):
        #The order of variables sorted in descending order by height of the attributes in
        #the factorization-tree
        attrs = sorted(self.jattrs, key= lambda x: self.dominance[x], reverse = True)
        #For ech tuples
        for t in tuples:
            #For each joining attribute starting from the ones at the bottom of the tree
            for attr in attrs:
                #Obtain the value of the attribute
                value = getattr(t, attr)
                #Obtain the child attributes of the given attributes
                fields = self.fields[attr]
                #Check whether a subtree for the current attribute with the given value
                #already exists otherwise create it
                if value in self.index[attr]:
                    node = self.index[attr][value]
                else:
                    #Creation of a new subtree
                    node = self.Fnode(attr, value)
                    #Attach it to the factorization-tree
                    self.index[attr][value] = node
                #For each child attribute...
                for field in fields:
                    #...obtain the child attribute value
                    value = getattr(t, field)
                    #Deposith the value in the correct leaf of the subtree
                    if field in self.jattrs:
                        #If it is a joining attribute then we obtain
                        #the corresponding subtree
                        other_node = self.index[field][value]
                        #Then we update the list of pointers in the tree
                        if field not in node:
                            node[field] = dict()
                        node[field][value] = other_node
                        #We update the list of parent pointers
                        other_node.parents.add(node)
                    else:
                        if field not in node:
                            node[field] = set()
                        node[field].add(value)

    #Count query
    def count(self):
        count = 1
        #For every root attribute
        for subtrees in self.view:
            partial = 0
            #For every subtree for the current attribute
            for value, subtree in subtrees.items():
                #Count the number of results represented in current subtree and
                #update the sum
                partial += subtree.count()
            count *= partial
        #Return
        return count

    #Given a named tuple returns the joining attribute having greater height in the
factorization tree
    def obtainJA(self, tuple):
        c = min([field for field in tuple._fields if field in self.jattrs], key = lambda x:
self.dominance[x])
        return c
```

```
    def onInsert(self, relation, tuple):
        if relation not in self.jrels or self.existingRecord(relation, tuple):
            return
        #Most dominant joining attribute
        jattr = self.obtainJA(tuple)
        #Value of the most dominant joining attribute
        key = getattr(tuple, jattr)
        #The other attribute in the tuple
        other_attr = tuple._fields[0] if tuple._fields[0] != jattr else tuple._fields[1]
        #The tuple's other attribute's value
        other_value = getattr(tuple, other_attr)
        #If a given subtree with the given joining attribute's value exists
        if key in self.index[jattr]:
            node = self.index[jattr][key]
            #If the other attribute is a joining attribute
            if other_attr in self.jattrs:
                #If there exists a subtree with value of the other joining attribute...
                if other_value in self.index[other_attr]:
                    #...we can avoid a join query and update directly the materialized view
                    other_node = self.index[other_attr][other_value]
                    #Update the child subtree's parent list
                    other_node.parents.add(node)
                    #update the subtree child list with the pointer to the child subtree
                    node[other_attr][other_value] = other_node
                    #Terminate here
                    return
            #Otherwise it is not a joining attribute so just deposit the value in the
correct leaf
            else:
                node[other_attr].add(other_value)
                #Terminate here
                return
        #If we reach this point we need to run the lightweight join query with the single-
record relation trick
        new_rel = (self.db[relation][0], {tuple}, self.db[relation][2])
        rels = [self.db[rel] if rel != relation else new_rel for rel in self.jrels]
        res = multiway_join(*rels)
        if len(res[1]) > 0:
            self.fill(res[1])

    def onDelete(self, relation, tuple):
        if relation not in self.jrels or self.nonExistingRecord(relation, tuple):
            return
        #Most dominant joining attribute
        jattr = self.obtainJA(tuple)
        #Dominant joining attribute value
        key = getattr(tuple, jattr)
        #Other attribute
        other_attr = tuple._fields[0] if tuple._fields[0] != jattr else tuple._fields[1]
        #Other attribute value
        other_value = getattr(tuple, other_attr)
        #If there exists no subtree with the given joining attribute value then we are done
        #as it means that the record play no role in the result set of the conjunctive
query.
        if key not in self.index[jattr]:
            return
        #If there exists a subtree...
        node = self.index[jattr][key]
        #..check if the other attribute is a non-joining attribute
        if other_attr not in self.jattrs:
            #...so just delete the value from the correct child of the subtree
            node[other_attr].discard(other_value)
        else:
```

```
            #Otherwise is a joining attribure so delete it from the list of pointers
            del node[other_attr][other_value]
        #Check if we emptied any leaf
        emptiedSomething = len(node[other_attr]) == 0
        if not(emptiedSomething):
            return
        #Queue of the nodes to visit
        toVisit = {node}
        #Blacklist (nodes to delete)
        toDelete = []
        #Obtain upward nodes to delete
        #1. Delete the all the links to che current subtree from parent nodes
        #2. If deleting the link to the current subtree empties any child of the parent
nodes
        #   put the parent in the black list as it needs to be deleted.
        while len(toVisit) > 0:
            n = toVisit.pop()
            for other_node in n.parents:
                #Delete link to the subtree from the parent's node
                del other_node[n.joining_attribute][n.value]
                #Check if we emptied any child from the parent
                if len(other_node[n.joining_attribute]) == 0:
                    #If yes, then we need to put the parent in the queue and in the
blacklist
                    toVisit.add(other_node)
                    toDelete.add(other_node)
        #Obtain downward nodes to delete
        #1. Delete all the links of a given node from its childs
        #2. Check if we empty any child's parent list
        #   as otherwise we need to put it in both the queue and blacklist
        toVisit = {node}
        while len(toVisit) > 0:
            n = toVisit.pop()
            for field in [key for key in n.keys() if key in self.jattrs]:
                for value, other_node in n[field].items():
                    other_node.parents.discard(n)
                    if len(other_node.parents) == 0:
                        toVisit.add(other_node)
                        toDelete.add(other_node)

        #Delete all the nodes in the blacklist
        for node in toDelete:
            del self.index[node.joining_attribute][node.value]
        #Delete the other original subtree referring to the deleted record's joining
attribute value
        if key in self.index[jattr]:
            del self.index[jattr][key]

    def result(self):
        return self.view
```

Question 2 (b) (c) – CYCLIC CONJUNCTIVE QUERIES

```
#Question 2(b), 2(c) - Cyclic Joins

from collections import defaultdict, namedtuple
from IncDB import IncDB
from FactorizedRepresentation_BinaryJoin import *
from functools import reduce
from operator import mul

#Cyclic joins representation is a hybrid between inverted index and factorization-trees
#Inverted index take care of the cyclic part of the result set of the conjunctive query (the
one involving all the join attributes)
#Factorization-trees take care of the acyclic part of the join if there is any

class Representation_ForCyclicJoins(IncDB):

    def __init__(self, database, jrels):
        IncDB.__init__(self, database, jrels)
        #The inverted index for the cyclic part of the result set
        self.index = dict()
        #Index for the subtree of the factorization-trees for the acyclic part
        #of the result set of the conjunctive query
        self.acyclic_fact_trees = dict()
        #A simple set for the cyclic part of the query
        self.cyclic_view = set()
        #Initialize inverted index
        for attr in self.jattrs:
            self.index[attr] = defaultdict(set)
        #This will contain the list of non-joining attributes paired indexed by the pairing
joining attribute
        dependency = defaultdict(list)
        #This is a list of "acyclic relations" (=relations with just a single joining
attribute)
        self.acyclic_rels = []
        for rel in jrels:
            if any(attr in self.otherAttrs for attr in database[rel][0]):
                self.acyclic_rels.append(rel)
                jattribute = database[rel][0][0] if database[rel][0][0] in self.jattrs else
database[rel][0][1]
                dependency[jattribute].append(rel)
        #For each joining attribute that has some non-joining attribute dependencies
        #build an instance of the factorization-tree to store that.
        for attr in dependency:
            self.acyclic_fact_trees[attr] = FactorizedRepresentation_BinaryJoin(database,
dependency[attr], attr)
        self.dependency = dependency
        self.cyclic_record = namedtuple('Record', self.jattrs)

    #Given a list of new result set tuples it updates the hybrid data structure
    def fill(self, tuples):
        for t in tuples:
            #Obtain the cyclic part of the current tuple
            sub_cyclic_record = self.cyclic_record(*[getattr(t, attr) for attr in
self.jattrs])
            #Updates the inverted index with pointers to the "cyclic portion" (the portion
containing joining attributes values)
            #of the current result tuple
            #Check if the "cyclic portion" of the current result tuple already does not
exist in the set
            if sub_cyclic_record not in self.cyclic_view:
```

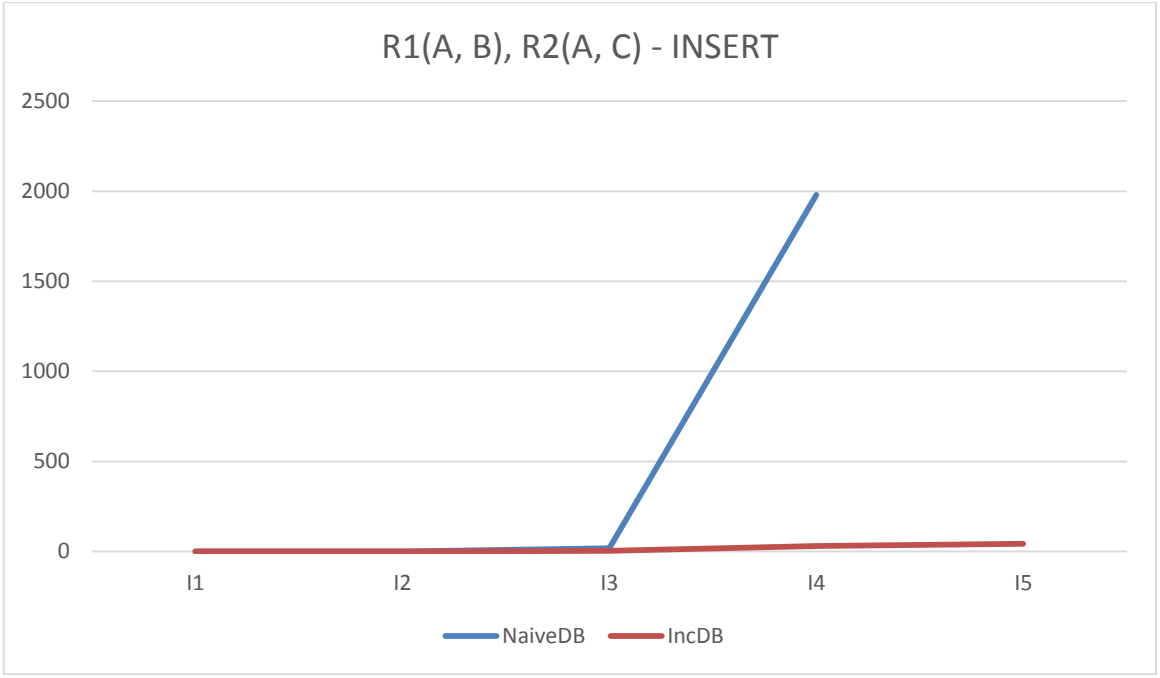
```
                #For each joining attribute update the (joining_attribute, joining_attribute
value) bucket
                #with pointer to the resultset tuple.
                for attr in self.jattrs:
                    key = getattr(t, attr)
                    index = self.index[attr][key]
                    index.add(sub_cyclic_record)
                self.cyclic_view.add(sub_cyclic_record)
            #Update the acyclic part of the query for every joining attribute
            #paired with non-joining attributes.
            #We delegate that to the factorization-tree implementation
            for rel in self.dependency:
                self.acyclic_fact_trees[rel].fill([t])


    def onInsert(self, relation, rec):
        if relation not in self.jrels or self.existingRecord(relation, rec):
            return
        #Check if the insert was done on a relation having a single joining attribute.
        if relation in self.acyclic_rels:
            #If that is the case...
            jattr = self.db[relation][0][0] if self.db[relation][0][0] in self.jattrs else
self.db[relation][0][1]
            value = getattr(rec, jattr)
            #...check before if there exists a bucket (jattr, value) in the inverted index
            if value in self.index[jattr]:
                #...there exists! So we can simply update the factorization-tree by
delegating that to
                #   its implementation!
                self.acyclic_fact_trees[jattr].onInsert(relation, rec)
                return

        #Otherwise either the relation has both the attributes being joining attributes
        #or there does not exist a bucket (jattr, value) in the inverted index
        #
        # anyway in both cases we need to run  a single-record relation join query
        new_rel = (self.db[relation][0], {rec}, self.db[relation][2])
        rels = [self.db[rel] if rel != relation else new_rel for rel in self.jrels]
        res = multiway_join(*rels)
        if len(res[1]) != 0:
            self.fill(res[1])

    #This purges the given result set "cyclic portion" tuples from the
    #hybrid data structure
    #deletedTuples is a list of the "cyclic portion" tuples to delete
    #purgeFactTree is a parameter which tells if we need to inspect the factorization-tree
(and update it if necessary)
    #              if we empty a bucket in the inverted index.
    def purge(self, deletedTuples, purgeFactTree = True):
        #For each tuple
        for t in deletedTuples:
            #For each joining attribute in the conjunctive query
            for attr in self.jattrs:
                v = getattr(t, attr)
                #Remove the tuple from the inverted index bucket (attr, v)
                self.index[attr][v].remove(t)
                #If we emptied the bucket...
                if len(self.index[attr][v]) == 0:
                    #...remove that from the inverted index
                    del self.index[attr][v]
                    #...check if we need to remove that from the factorization-tree as well
                    #   if the current attribute is part of factorization-tree
                    if purgeFactTree and attr in self.acyclic_fact_trees:
                        del self.acyclic_fact_trees[attr].view[v]
```

```
            #Remove the tuple from the result set
            self.cyclic_view.remove(t)


    def onDelete(self, relation, rec):
        if relation not in self.jrels or self.nonExistingRecord(relation, rec):
            return
        #Check if the relation of the deleted tuple is cyclic (= attributes of the relations
are both
        #joining attributes)
        if relation not in self.acyclic_rels:
            a,b = rec._fields
            x,y = rec
            index = self.index
            #Compute the intersection between the buckets in the result set
            deletedTuples = index[a][x].intersection(index[b][y])
            #Remove the tuples from the hybrid data structure
            self.purge(deletedTuples)
        #Otherwise if the relation has non-joining attributes..
        else:
            jattr = self.db[relation][0][0] if self.db[relation][0][0] in self.jattrs else
self.db[relation][0][1]
            #...we are operating over factorization-trees so we delegate the deletion of
that to the
            #factorization-tree implementation
            self.acyclic_fact_trees[jattr].onDelete(relation, rec)
            v = getattr(rec, jattr)
            #Check if we have emptied the subtree referring to the deleted record...
            if v not in self.acyclic_fact_trees[jattr].view:
                #...in which case we need to purge all the records falling in the inverted
index bucket (jattr, v)
                deletedTuples = self.index[jattr][v].copy()
                self.purge(deletedTuples, False)

    #Count over the hybrid inverted-index factorization-tree representation
    def count(self):
        #1. Count over factorization-tree by creating a temporary 2-level hash-table of the
form
        #   attribute -> value -> count
        acyclic_count = defaultdict(dict)
        #For every joining attribute involved in some relation with another
        #non-joining attribute
        for attr in self.dependency:
            #Obtain the factorization-tree for that attribute
            fact_tree =  self.acyclic_fact_trees[attr]
            #For every subtree...
            for value, node in fact_tree.view.items():
                #...multiply the number of the child attributes
                count_value = [ len(values) for _, values in node.items()]
                count_value = reduce(mul, count_value)
                #Update the temporary hash-table bucket (attr, value)
                acyclic_count[attr][value] = count_value
        #Iterate over the "cyclic" result set entry and count the number of combinations
        #having the entry itself as "prefix"
        count = 0
        for cyclic_result in self.cyclic_view:
            partial = 1
            #For each joining attribute involved in some relation with another
            #non-joining attribute
            for attr in self.dependency:
                value = getattr(cyclic_result, attr)
                #Update the partial count
                partial *= value
            #Update the global count
```

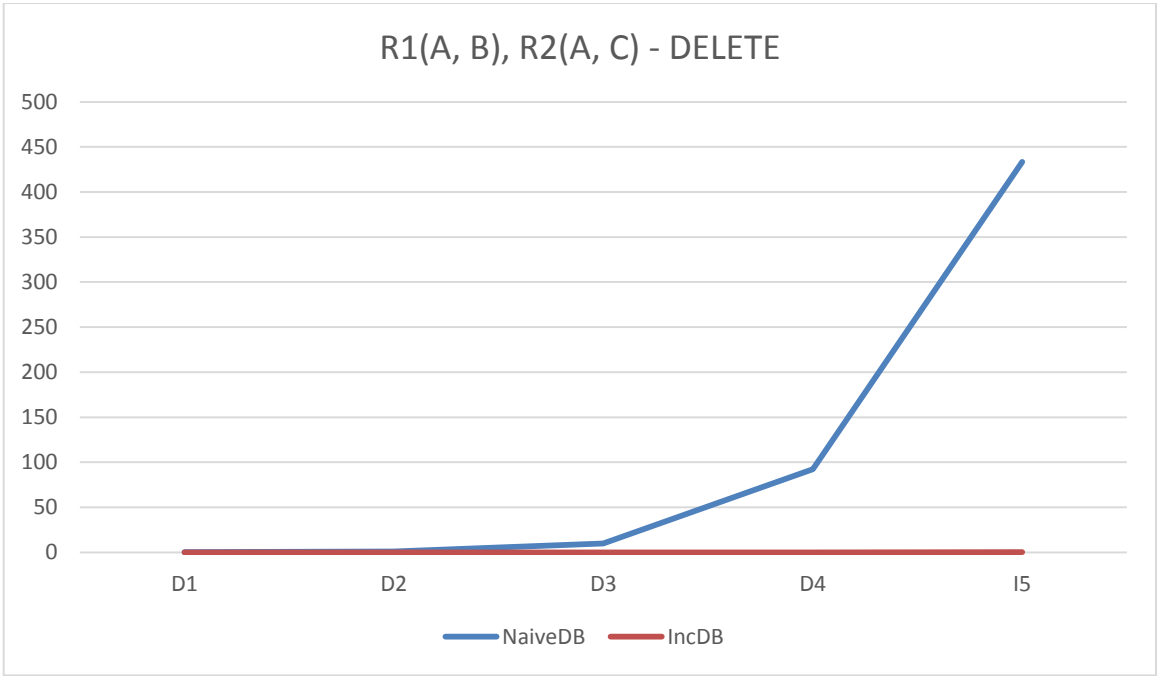
```
        count += partial
    return count
```

Question 3

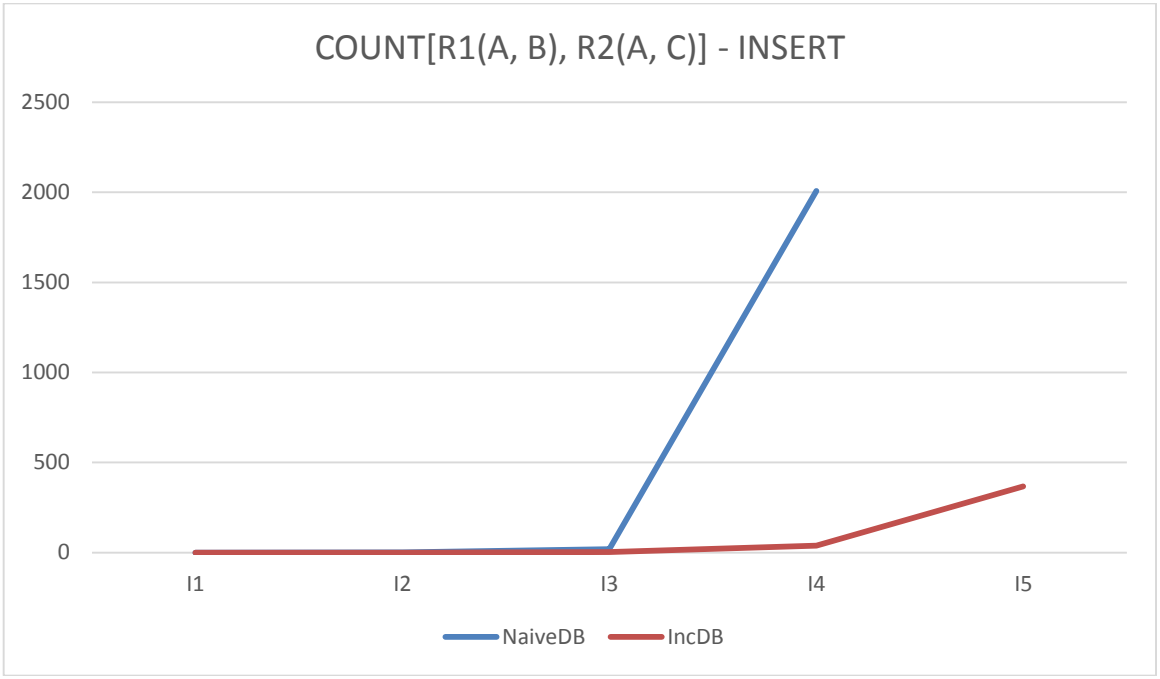
Question 3 (a)



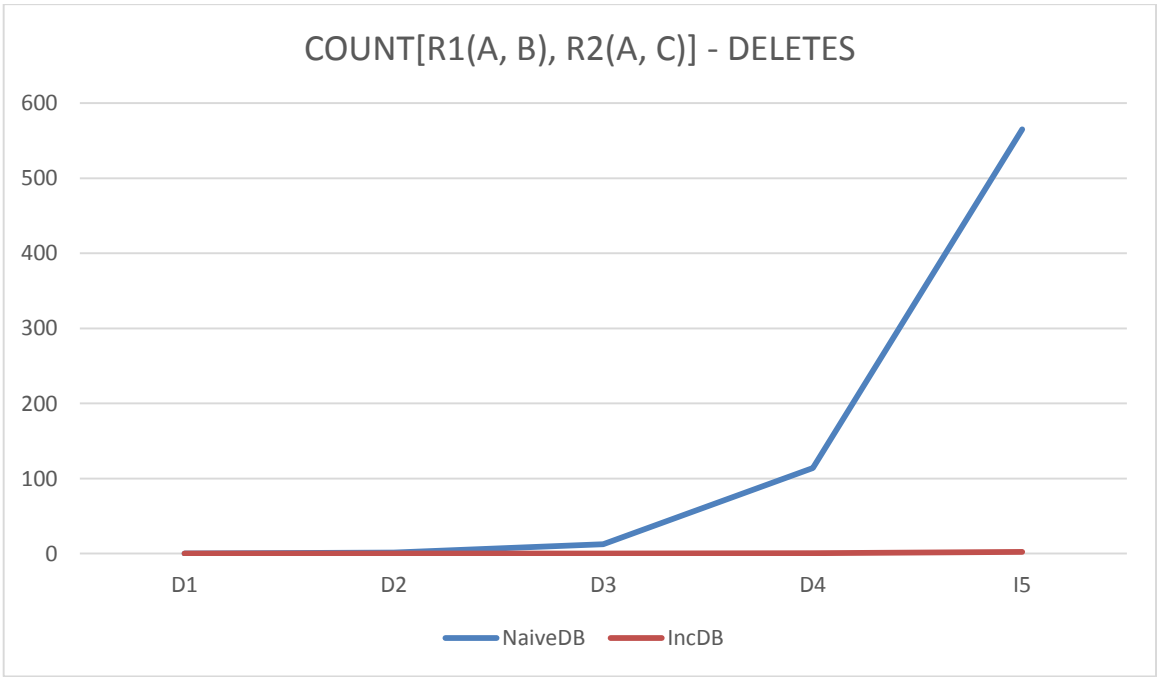
IncDB clearly outperforms NaiveDB in the above scenario, my intuition is that the insertions executed are of the kind that IncDB does not need to run an equi-join query for updating the table of the result.
NaiveDB on the other side must recompute the result of the query from scratch and it is not able to process the dump “I5” in a reasonable time (the timeout set to 3600 seconds)



We got to remember that IncDB never and never executes a join query for updating the materialized view hence the gap between IncDB and NaiveDB visible in the above plot.

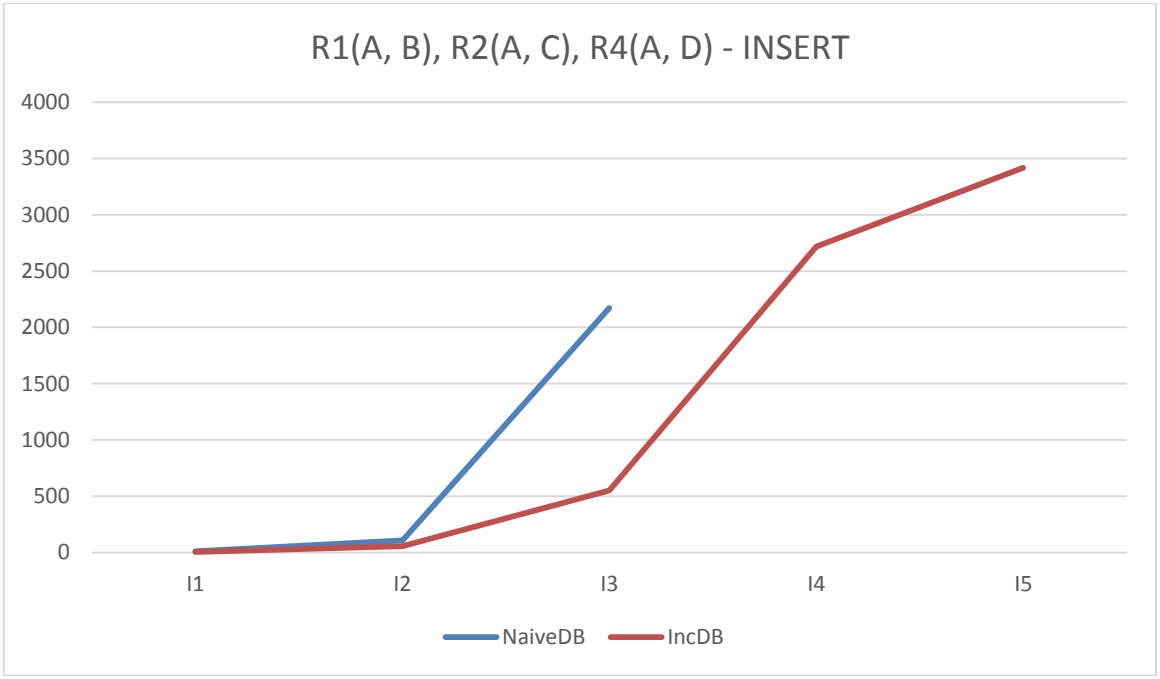


NaiveDB seems to be performing equally under count queries, this is interesting, I guess it is because as NaiveDB is implemented through a set presents a very low number of cache misses which does not add excessive overhead in computing the COUNT query.
IncDB on the other side still outperforms NaiveDB but the time required to compute a COUNT query over inserts increases noticeably compared with plain simple inserts.
A possible explanation is that other than having a big number of results (653889) it shows the downside of tree-like data structures: they are not cache-oblivious meaning that probably the slowdown is given by the number of cache misses as it is a very sparse data-structure.

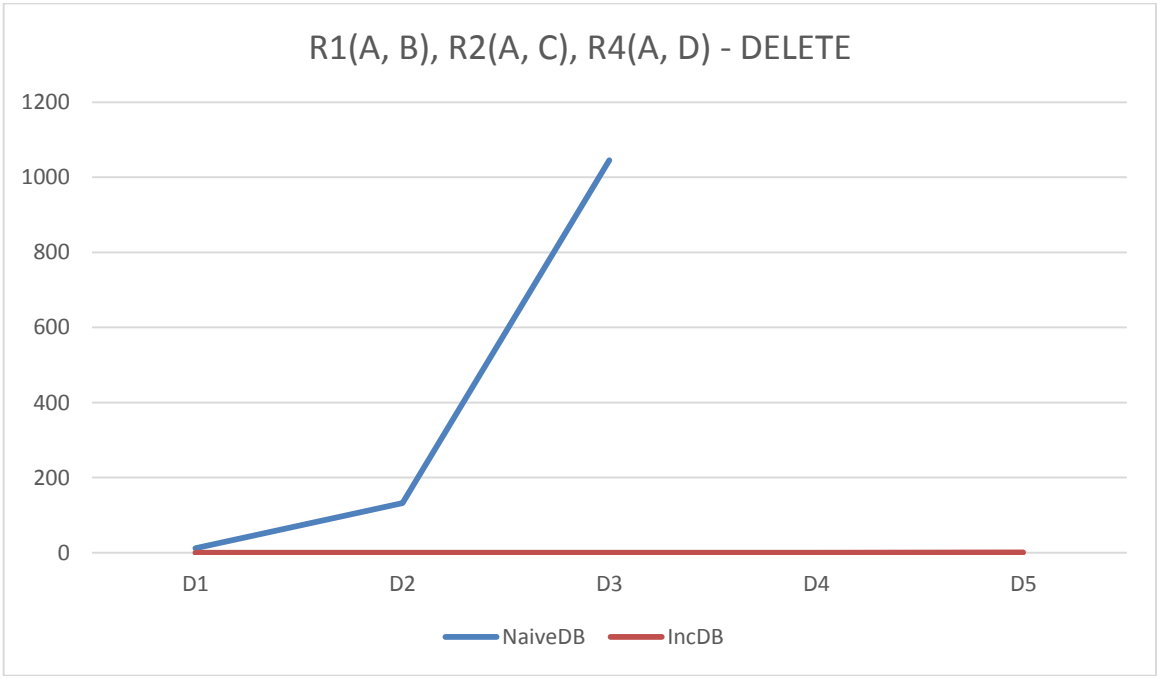


I don’t think there is anything to say about the above plot, it just shows again that bit of cleverness present in IncDB that shows in a deletions-heavy scenario.

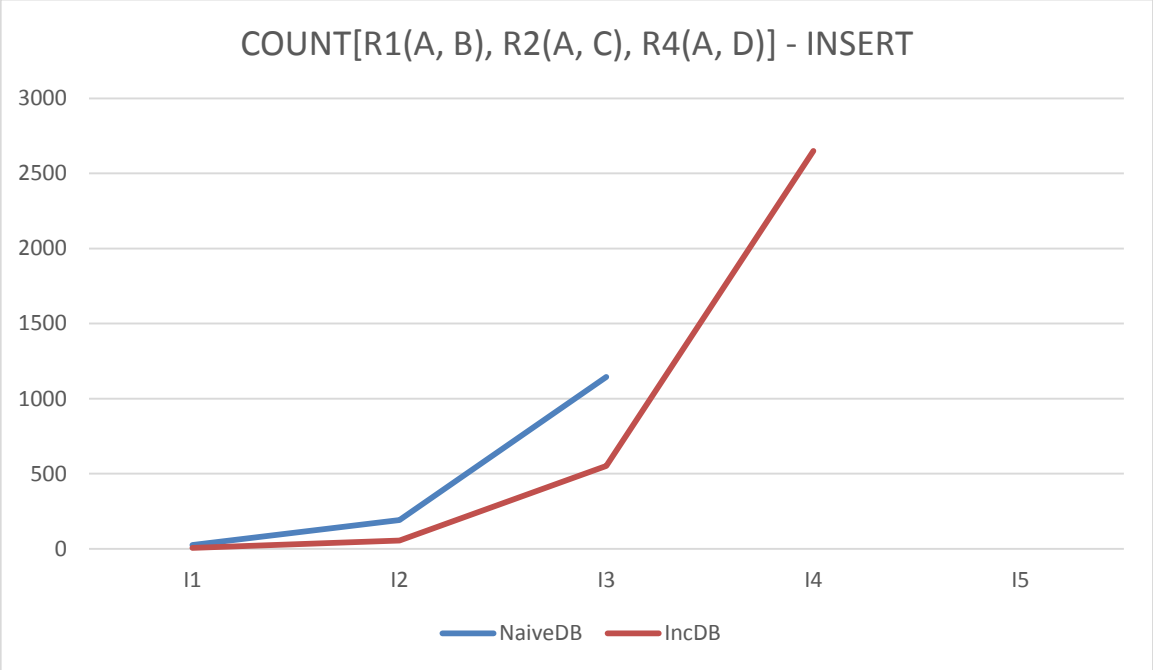
Question 3 (b)



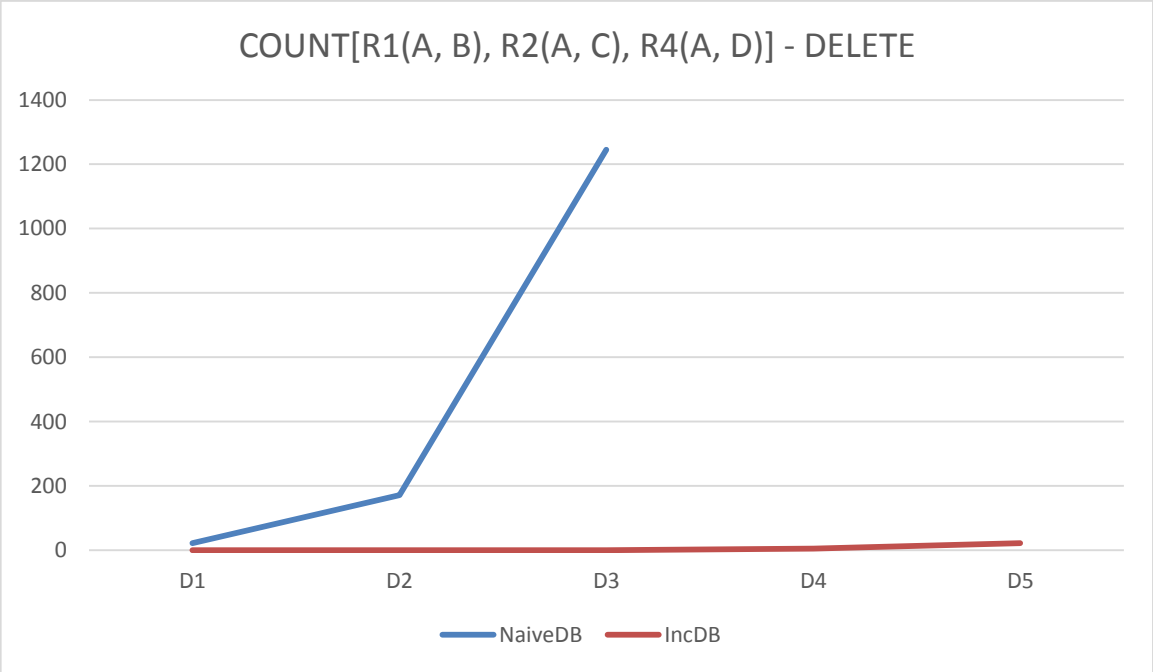
IncDB still outperforms NaiveDB with the latter not being able to process the last 2 files but I think that the plot is interesting as IncDB is not as performant as it was under the conjunctive query present in the first question.
The reason is given by the fact that probably the insertions executed forced IncDB to execute a good number of lightweight join queries to obtain the results entries to add to the materialized view.
Here it shows a bad design decision of ours in the choice of the join algorithm as we opted for the Hash-Join algorithm.



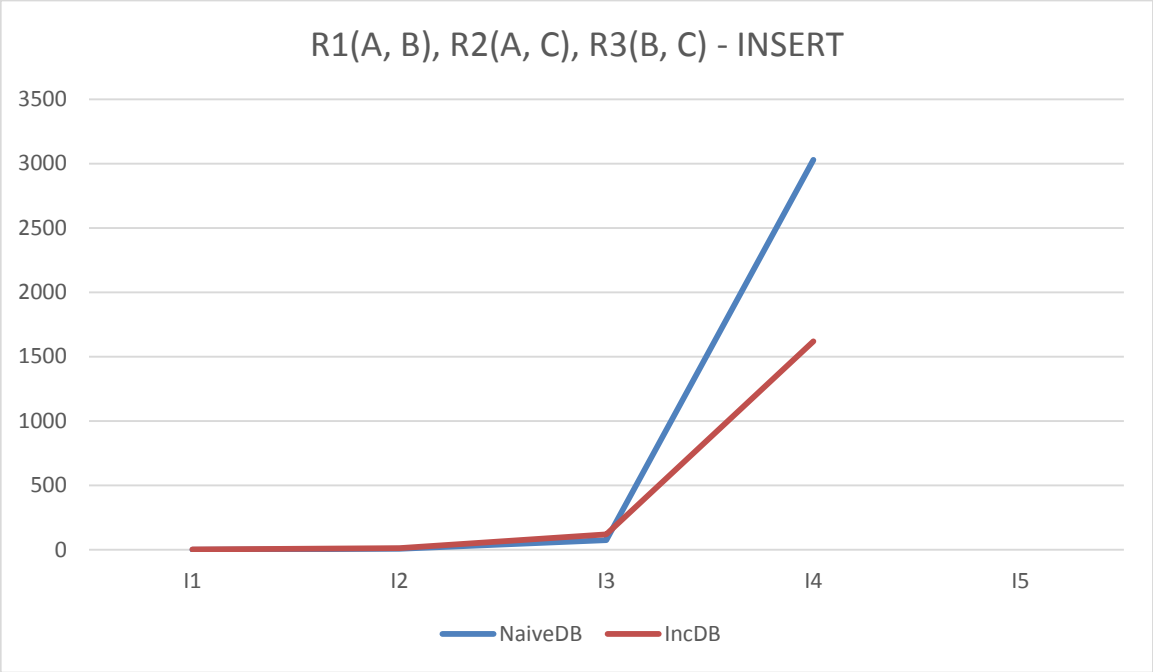
If we had any doubt about the performance of IncDB over deletions, they are vanished with this plot.
The tables R1, R2 and R4 probably make the join prohibitive for NaiveDB as it has to recompute the whole result set from scratch, IncDB on the other side operates directly on the materialized view and can simply return it to the user.



The news there is that IncDB is not able to process a file (I5), that is probably due to the fact that probably the factorization-tree is not dense (the set of insertions probably does not compress well as there are subtrees which leafs hold single values) other than the problem highlighted before regarding the lightweight joins queries that need to be run if a given joining attribute's value is not present in the factorization-tree.
The above plot helps building the case for batch updates of the materialized view.
NaiveDB's performance is no news, it is not able to process I4 and I5 sets of insertions.



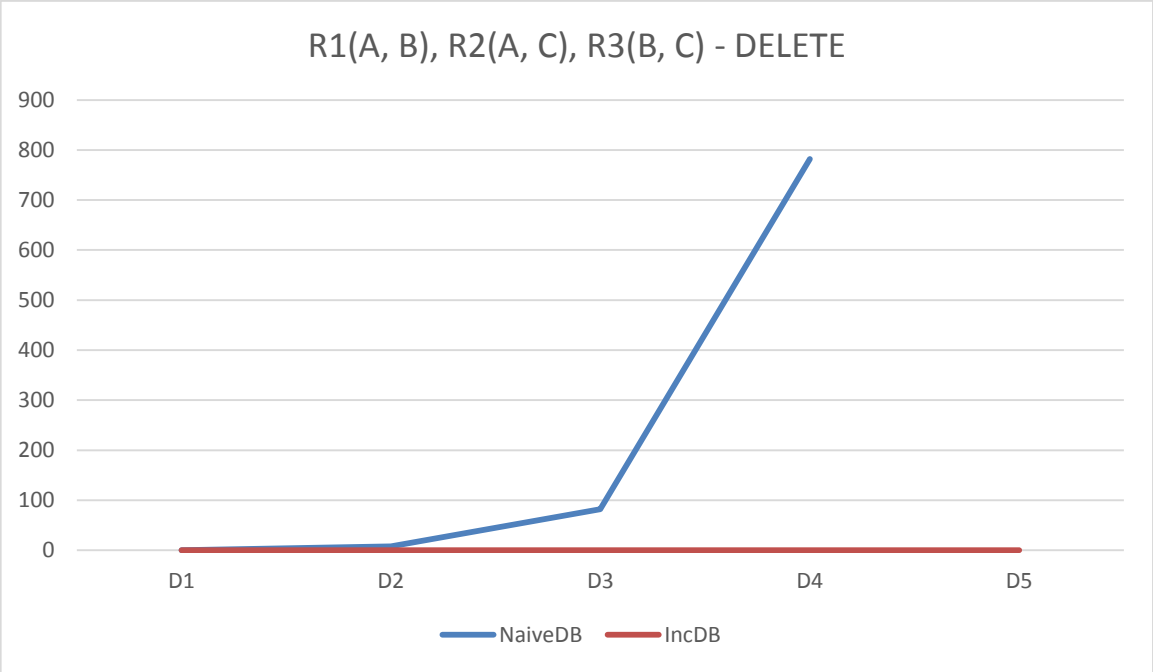
I don’t think there is anything to say, we have talked extensively about the good deletions performance of IncDB in the previous plots.



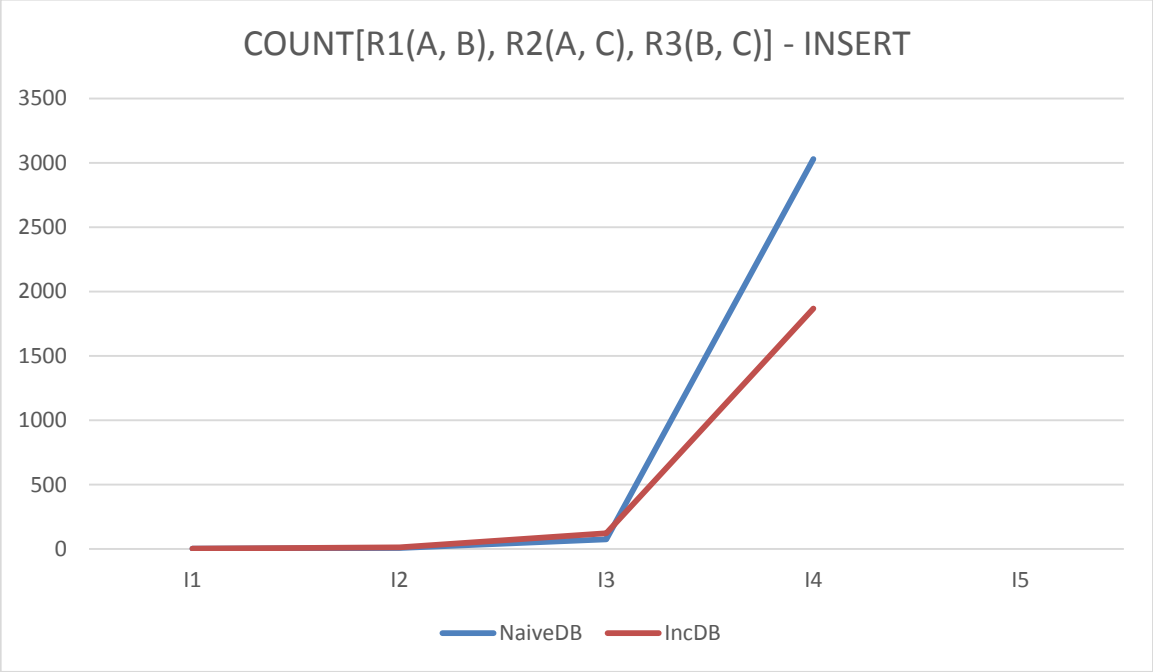
Before diving into the analysis I want to remember that the conjunctive query "R1(A, B), R2(A, C), R3(B, C)" is a fully cyclic join meaning that at EVERY insertion incDB needs to run a lightweight join query for the new entries to put in the materialized view.
While we outperform NaiveDB except for small number of insertions (I1, I2, I3) which is probably due to the fact the tables are sufficiently small that sort-merge join returns early the news is that we are not able to process the sets of insertions I5.
The shortcomings in IncDB are two:
- We do not update the materialized view in batch as we do that at every query.
- IncDB seriously needs a worst-case optimal join algorithm. [5]

We will continue the discussion about the shortcomings after this question as we propose two improvements for IncDB.

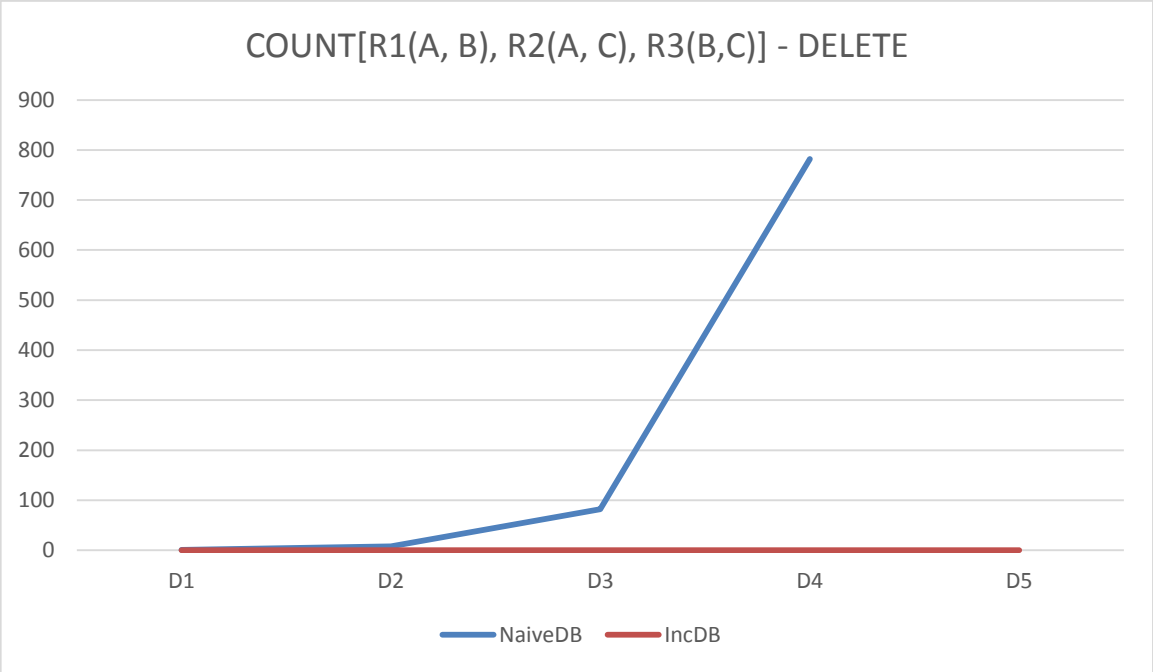
[5] Veldhuizen, Todd L. "Leapfrog triejoin: a worst-case optimal join algorithm." *arXiv preprint arXiv:1210.0481* (2012).



No news about this plot plus the result set being freakishly small helps.



We feel that what we said about the performance of IncDB under conjunctive query "R1(A, B), R2(A, C), R3(B, C)" still applies there.
Obviously IncDB while outperforming NaiveDB could make use of some improvements.



No news about this plot.

Lessons learned in designing and implementing IncDB

Overall we managed to develop a small db engine that outperforms NaiveDB, we accomplished this with the use of factorization-trees that provide a compressed version of the result set but more important with the lightweight join query trick:

$$(R_1 \cup t) \bowtie R_2 = R_1 \bowtie R_2 \cup (t_{R_1} \bowtie R_2)$$

These two tricks are agnostic of the chosen join algorithm, which I thought was the point of the question: agnosticism from the chosen join algorithm.

Though I am not satisfied overall as the experiments hint us that we can do better, I made two bad design decisions:

- Choice of join algorithm (Hash-Join)
- Sequential update of the materialized view

Hash-Join algorithm unfortunately is linear in the size of the relations having complexity $\theta(n)$.
We could have done lot better by employing the leapfrog trie-join algorithm [6] which has been proved to be worst-case optimal.
Under the leapfrog trie-join algorithm the tables are represented through factorization-trees so some care is needed in order to come up with good factorization-trees with retrospect a given conjunctive query.

We could have also implemented batch updates instead of updating the materialized view upon every insertion.
So suppose we have three relations R_1, R_2, R_3, we denote with t_i the sets of tuples of the relation R_i.
We can then obtain the set of the new entries in the result set:

$$(t_1 \bowtie R_2 \bowtie R_3) \cup (t_2 \bowtie R_1 \bowtie R_3) \cup (t_3 \bowtie R_1 \bowtie R_2)$$

So instead of running one light-weight join query upon every insertion we can do “batch update” and run instead a number of join queries equivalent to the number of relations in the conjunctive query to be materialized.
This would have definitely speeded up thing in the instance of the fully cyclic join query.

[6] Veldhuizen, Todd L. "Leapfrog triejoin: a worst-case optimal join algorithm." *arXiv preprint arXiv:1210.0481* (2012).